A/UX Command Reference

Section 1(A–F)

Release 3.0

Limited Warranty on Media and Replacement

If you discover physical defects in the manuals distributed with an Apple product or in the media on which a software product is distributed, Apple will replace the media or manuals at no charge to you, provided you return the item to be replaced with proof of purchase to Apple or an authorized Apple dealer during the 90-day period after you purchased the software. In addition, Apple will replace damaged software media and manuals for as long as the software product is included in Apple's Media Exchange Program. While not an upgrade or update method, this program offers additional protection for up to two years or more from the date of your original purchase. See your authorized Apple dealer for program coverage and details. In some countries the replacement period may be different; check with your authorized Apple dealer.

ALL IMPLIED WARRANTIES ON THE MEDIA AND MANUALS, INCLUDING IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE, ARE LIMITED IN DURATION TO NINETY (90) DAYS FROM THE DATE OF THE ORIGINAL RETAIL PURCHASE OF THIS PRODUCT.

Even though Apple has tested the software and reviewed the documentation, **APPLE MAKES NO WARRANTY OR REPRESENTATION, EITHER EXPRESS, OR IMPLIED, WITH RESPECT TO SOFTWARE, ITS QUALITY, PERFORMANCE, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. AS A RESULT, THIS SOFTWARE IS SOLD "AS IS," AND YOU, THE PURCHASER, ARE ASSUMING THE ENTIRE RISK AS TO ITS QUALITY AND PERFORMANCE.**

IN NO EVENT WILL APPLE BE LIABLE FOR DIRECT, INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES RESULTING FROM ANY DEFECT IN THE SOFTWARE OR ITS DOCUMENTATION, even if advised of the possibility of such damages. In particular, Apple shall have no liability for any programs or data stored in or used with Apple products, including the costs of recovering such programs or data.

THE WARRANTY AND REMEDIES SET FORTH ABOVE ARE EXCLUSIVE AND IN LIEU OF ALL OTHERS, ORAL OR WRITTEN, EXPRESS, OR IMPLIED. No Apple dealer, agent, or employee is authorized to make any modification, extension, or addition to this warranty.

Some states do not allow the exclusion or limitation of implied warranties or liability for incidental or consequential damages, so the above limitation or exclusion may not apply to you. This warranty gives you specific legal rights, and you may also have other rights which vary from state to state.

Apple Computer, Inc.

Apple Computer, Inc.
20525 Mariani Avenue
Cupertino, CA 95014-6299
(408) 996-1010

Simultaneously published in the United States and Canada.

A/UX Command Reference

Contents

About This Manual

This manual is one of three primary manuals in the set of A/UX reference manuals. *A/UX Command Reference*, *A/UX Programmer's Reference*, and *A/UX System Administrator's Reference* contain information about most of the provisions of A/UX, such as its commands, its library routines, its system calls, and its file formats.

These reference manuals constitute a compact encyclopedia of A/UX information. As in an encyclopedia, the information is subdivided into subdocuments, or "manual pages." The information in each manual-page subdocument adheres to a distinctive presentation format. For example, information about command syntax is consistently presented under the heading "Synopsis." (This format is described in detail later in this preface.)

Because most of us need occasional reminders regarding the order and kind of arguments that can accompany a command, the information in the "Synopsis" and "Arguments" sections is presented for use by users at all levels. However, the information in the "Description" section is often written for more advanced users; novices most likely will not be able to learn about the provisions of A/UX from these reference manuals alone.

Because these reference manuals are not intended to be tutorials or learning guides, they should not be the first A/UX books you read. If you are new to A/UX or are unfamiliar with a specific functional area (such as the Macintosh Finder), you should first read *A/UX Essentials* and the other A/UX user guides. After you have worked with A/UX, the reference manuals can help you understand new features or refresh your memory about features you already know.

Manual pages: a standard for presenting information

The headings conventionally used in the manual pages have virtually become an industry standard for reference documents. Furthermore, the way that this large collection of subdocuments is conventionally organized into sections and books is also something of a standard.

Despite the standardization, locating specific information within this large body of documentation can often be difficult. First you must locate the correct manual page. Once you have the correct manual page, you can usually go

directly to the correct subsection.

To help you locate information, you should read the next section, which explains several means of finding the information you need.

To help you learn to use these books more effectively, other sections in this preface describe the presentation standards that are being used. Some of these are organizational standards that apply at the book and section level. Other conventions and content standards apply within the scope of each manual page, such as the use of standard subheadings and the conventional use of certain fonts and text styles.

Note that the most durable standards have been the standards that apply to the organization and primary headings of each manual page. Of course there are areas in which the A/UX reference books are exceptional, particularly in their more regular use of headings. These books also deviate from industry standards in a few typographic and style areas, which are described later in this preface. For example, the Courier font is used consistently to represent text that is displayed in a terminal window or entered as part of a command line. Other UNIX® books often use boldface type to represent such text.

There has been more instability with respect to how the manual pages are collected into sections and books. For more detailed information, see ''Previous Organization of Sections into Books'' later in this preface.

Locating information in the reference manuals

You can locate information in the reference manuals by using one of the following tools:

- Table of contents. Each reference manual contains one general table of contents for the entire manual. Located at the beginning of each new section of manual pages is a detailed table of contents. (If a section must span from one binder to another, a tailored table of contents is provided for each of the subdivisions.) The general table of contents lists the sections covered in the complete manual. The detailed table of contents lists the manual pages contained within one section (or section subdivision) along with a brief description of the A/UX provision that is covered in each manual page.
- Query commands. The `man`, `whatis`, and `apropos` commands display on-screen all the information contained in a manual page or just the information in the ''Name'' section of one or more manual pages that

satisfy a search criterion. The next sections tells you how to use the on-line versions of the manual pages.

- *A/UX Reference Summary and Index.* This separate manual is considered part of the A/UX set of reference manuals, but it is not a ''standard'' resource like the other reference materials. Its primary purpose is to help you locate the correct manual page to refer to in other books. From its summaries, you might also occasionally find all the information you required. It contains the following subsections:
  - ''Commands by function.'' This subsection classifies the A/UX user and system administrator commands by the general or most important function each performs. The summary gives you a broader view of the commands that are available and the context in which each is often used. Each command is mentioned just once in this listing.
  - ''Command synopses.'' This subsection is a compact collection of syntax descriptions for all of the commands in *A/UX Command Reference* and in *A/UX System Administrator's Reference*. It may help you find the syntax of commands more quickly when the syntax is all you need.
  - ''Index.'' The index lists key terms associated with A/UX subroutines and commands. These key terms can help you locate the manual page you need when you don't know if such a keyword-related command or subroutine exists.

The index provided in *A/UX Reference Summary and Index* is designed to be more compact and easier to use than the more industry-standard permuted index, which indiscriminately indexes manual pages under each of the words found in their ''Name'' sections.

The manual pages listed in the index portion *A/UX Reference Summary and Index* are indexed under more than one entry; for example, `lorder`(1) is included under ''archive files,'' ''sorting,'' and ''cross-references.'' By using this type of index, you are more likely to find the reference you are looking for on the first try.

Using the on-line documentation

In addition to the paper documentation in the reference manuals, A/UX provides several ways to search and read the contents of each manual page from your A/UX system. An advantage to the on-line version of the documentation is that the computer performs the work of filtering out (or skipping) all the manual

pages other than the one you specifically queried. The only prerequisite is that you already know its name (or a proper search string). However, you don't have to know how manual pages are organized by section numbers and by book titles.

To display a manual page on your screen, enter the `man` command followed by the name of the manual page you want to see. For example, to display the manual page for the `cat` command, including its description, syntax, options, and other pertinent information, you would enter

```
man cat
```

After the first screen of the text of a manual page appears, you can display subsequent screens of the text with each press of the SPACE BAR, until you reach the end of the man page. To display subsequent text one line at a time, press RETURN instead of the SPACE BAR. By pressing Q, you can quit the `man` command before viewing all of the manual page.

To display the descriptive information in the ''Name'' section of any manual page, enter the `whatis` command followed by the name of the provision you want described. In the following example, the command prompt is the percent sign, and the provision that is being queried is the `ls` command:

```
% whatis ls
ls(1)              - lists the contents of a directory
% ■
```

To display a list of all manual pages whose ''Name'' sections contain a given keyword or string, enter the `apropos` command followed by a search word or search string enclosed in double quote characters. The names of A/UX provisions are listed on separate lines along with the descriptive information in the ''Name'' section of the manual page that describes those provisions. Sometimes several A/UX provisions are listed on the same line. In those cases, several A/UX provisions are described on a single manual page. You can tell which of these names is the formal name for the manual page because it will be followed by parentheses and an enclosed section number. In the following example, the command prompt is the percent sign, and the A/UX provisions that are queried are those which are described in manual pages whose ''Name'' section contains the word ''tape'':

```
% apropos tape
mt(1)           - magnetic tape manipulating program
frec(1M)        - recover files from a backup tape
mtio(7)         - interface conventions for magnetic tape devices
tc(7)           - Apple Tape Backup 40SC device driver
% ■
```

These documentation query commands are described more fully in the manual pages `man`(1), `whatis`(1), and `apropos`(1) in *A/UX Command Reference*.

Book- and section-level presentation standards

Customarily, three books are used to house three collections of manual pages that are of concern to three different audiences:

- *A/UX Command Reference* is intended for users with normal file and device access privileges.
- *A/UX System Administrator's Reference* is intended for system administrators or equivalent users with unlimited device and file access privileges.
- *A/UX Programmer's Reference* is intended for programmers.

These books are further divided into sections, each of which contains a set of manual pages in alphabetical order. The standard sections and the audiences they serve are as follows:

- For users with normal access privileges, Section 1 and Section 6 describe utility and game commands.
- For users with unlimited access privileges, Section 1M and Section 8 describe system maintenance commands.
- For programmers, Section 2 describes system calls, Section 3 describes library routines, Section 4 describes file formats, Section 5 describes miscellaneous A/UX provisions, and Section 7 describes drivers and interfaces for devices.

While most of the manual pages describe an A/UX provision of some sort, there is one important exception per section: The first manual page in Sections 1, 1M, 2, 3, 4, 5, 6, 7 and 8 has the same name, `intro`. The `intro` manual pages do not describe a command or other provision of A/UX. Instead, they serve as an introduction to the rest of the manual pages in the section, providing section-

specific information and conventions. (These section-introduction manual pages are also exceptions in terms of the normal alphabetical arrangement of manual pages inside sections.)

For example, the manual page `intro`(2) introduces you to return values and provides an exhaustive list of error code values and their associated error strings. In the rest of the Section 2 manual pages, the error codes are mentioned briefly or merely listed, without detailed explanations.

More advanced readers will probably have occasion to use more than one of the reference manuals. For example, manual pages in the *A/UX Programmer's Reference* frequently make references to manual pages in sections contained in the other two primary reference manuals.

More information about the organization of the reference books is given later in this preface in "Current Organization of Sections into Books."

How manual-page information is presented

The name of the manual page normally appears in both upper corners of each physical page. Some manual pages describe several routines or commands. For example, `chown` and `chgrp` are both described in a manual page with the primary name `chown`(1) at the upper corners. If you turn to the page `chgrp`(1), you find a reference to `chown`(1). (These cross-reference pages are included only in *A/UX Command Reference* and *A/UX System Administrator's Reference*.) However, if you enter the command `man chgrp`, the extended-coverage `chown`(1) manual page is displayed automatically.

All of the manual pages have a common format that uses the following subheadings. For the most part, the same kind of information appears under each of these subheadings. However, for manual pages that describe different kinds of A/UX provisions, the information under the same heading may differ. So, for example, the heading "Synopsis" contains syntax illustrations for Sections 1, 1M, and 8, but contains C declaration statements for Sections 2 and 3.

NAME
This section lists the names of the commands, programming routines, or other A/UX provisions that are described in the manual page. A succinct statement of their purpose is also provided.

SYNOPSIS
This section provides the syntax of a command or the data-type declarations associated with a programming routine.

ARGUMENTS
This section lists and describes the command options and arguments that can follow the command name on the command line.

DESCRIPTION
This section describes in detail the usage of a particular command or programming provision.

EXAMPLES
This section offers representative command lines that illustrate various uses of a command.

STATUS MESSAGES AND VALUES
This section describes possible error outcomes and, when applicable, possible success outcomes. For commands, exit values are not usually described if the command produces the customary zero exit value for success and a nonzero exit value for failure. For programming routines, the return value from a function is often an indication of completion status. In such cases, the return value is normally discussed in the ''Description'' section as well as in this section.

WARNINGS
This section describes possible usage scenarios that can damage the file system or file integrity or that produce results you would not normally anticipate.

LIMITATIONS
This section describes how the performance of a command or routine could become unreliable, or areas of functionality that an A/UX provision does not address.

NOTES
This section provides miscellaneous information regarding a command or routine, such as author or copyright information.

FILES
This section lists any files needed by the command, along with a brief description that identifies it as a file, directory, or link.

SEE ALSO
This section provides a list of references to related information.

Visual conventions for the A/UX reference manuals

A/UX books follow specific styling conventions. For example, words that require special emphasis appear in specific fonts or styles. This section describes

the conventions used in all the A/UX reference books.

Keys and key combinations

Certain keys on the keyboard have special names. These modifier and character keys, often used in combination with other keys, perform various functions. In this book, the names of these keys appear in the format of an initial capital letter followed by small capital letters.

Here is a list of the most common key names:

CAPS LOCK	ENTER	SHIFT
COMMAND	ESCAPE	SPACE BAR
CONTROL	OPTION	TAB
DELETE	RETURN	

Sometimes two or more key names are joined by hyphens. The hyphens indicate that you press these keys simultaneously to perform a specific function. For example,

Press CONTROL-K

means "While holding down the CONTROL key, press the K key."

Terminology

In A/UX manuals, a certain term can represent a specific set of actions. For example, the word "enter" indicates that you type a series of characters, then press the RETURN key. The instruction

Enter `whoami`.

means "Type `whoami`, then press the RETURN key." (If you entered this text at a command prompt, the system would respond by displaying your current account name.)

Here is a list of common terms and their corresponding actions.

Term	Action
Click	Press and then immediately release the mouse button.
Choose	Activate a command that appears in a menu. To choose a command from a pull-down menu, position the pointer on the menu title and, while holding down the mouse button, slide the mouse toward you until the command is highlighted. Then release the mouse button.
Drag	Position the pointer on an icon, press and hold down the mouse button while moving the mouse so that the icon moves to the desired position, and then release the mouse button.
Enter	Type the series of characters indicated, then press the RETURN key.
Press	Press one key only. (Do not press the RETURN key afterward.)
Select	To select an icon, position the mouse pointer on the item, then click (see "Click," above). To select text, use a drag-style operation (see "Drag," above). When selecting a range of text, the drag operation highlights the text from the starting point over and across lines to the final position of the pointer when the mouse button was released.
Type	Type the series of characters indicated, without pressing the RETURN key afterward.

The Courier font

Throughout the A/UX reference manuals, words that appear on the screen or that you must type exactly as shown are in the Courier font.

Here's an example:

Type `date` on the command line and press RETURN.

This instruction means that you should type the word ''date'' exactly as shown, then press the RETURN key.

After you press RETURN, text such as this will appear on the screen:

```
Fri Nov  1 11:15:43 PST 1991
```

In this case, the Courier font is used to represent exactly what appears on the screen.

All A/UX manual page names are shown in the Courier font. For example, `ls`(1) indicates that `ls` is the name of a manual page that occurs in Section 1. More information about the use of the Courier font in manual pages is given in ''Styling of A/UX Command Elements'' and in ''Styling of Cross-References to Manual Pages'' later in this preface.

Font styles

Italics are used to indicate that a word or set of words is a placeholder for part of a command line. Here is a sample command syntax illustration:

`cat` *file*

The italicized term *file* is a placeholder for the name of a file. If you wanted to display the contents of a file named `Elvis`, you would type the filename `Elvis` in place of *file*. In other words, you would enter

```
cat Elvis
```

Styling of A/UX command elements

A/UX commands are entered in accordance with their command syntax. A typical A/UX command line includes the command name first, followed by options and arguments. For example, here is an illustration of the syntax for the `wc` command:

`wc [-l] [-w]` *file...*

In this syntax illustration, `wc` is the command, `-l` and `-w` are options, and *file* is an argument.

A ''command option'' modifies the action of a command, often by changing its mode of operation (such as read mode or write mode).

An ''argument'' is any element that follows the command name. Command arguments other than command options typically specify the objects upon which the command should act. You often supply the names of files that you want a command to process, so *file* is frequently the last element in syntax illustrations.

Brackets and ellipsis characters in a syntax illustration should be considered part of a syntax notation. This is represented by the use of body font instead of Courier for these characters. Their font treatment tells you that you are not supposed to type these characters as part of the command line. Their meaning as a syntax notation is described next.

The brackets enclose an optional item or a group of optional items. If an optional item has constituent parts that are also optional, these parts are themselves enclosed in brackets, as in this syntax illustration:

`lpr [-i [`*numcols*`]]`

This syntax illustration shows that the indent (`-i`) command option can be followed by the number of columns to indent the printed page. It also shows that you can omit the number of columns; if you do, the `lpr` command uses the default indent value.

An ellipsis (...) follows an argument that can be repeated any number of times on a command line. If the ellipsis follows a bracketed group of items, the group of items can be repeated any number of times on the command line.

When command options are mutually exclusive, they cannot both be specified at the same time. In such cases, more than one syntax illustration is usually provided:

`pax -r[`*other-option-for-archive-reading*`]...`
`pax -w[`*other-option-for-archive-writing*`]...`

Outside of syntax illustrations, command options are shown with a leading hyphen also in the Courier font. When you supply multiple command options in an actual command line, only one leading hyphen is normally required. For example the following command line contains two options, `-r` and `-f`:

```
pax -rf /dev/rfloppy0
```

In the example, the `-f` option (pronounced ''minus f'') is entered without its own hyphen, even though when mentioned in running text it appears with a leading hyphen.

Styling of cross-references to manual pages

The manual pages are organized primarily in terms of sections, and secondarily in terms of books for different audiences. The standard A/UX cross-reference notation leaves out the book title, but refers to the section designation:

item(*section*)

where *item* is the name of the command, subroutine, or other A/UX provision, and *section* is the section where the manual page resides.

For example,

`cat`(1)

refers to the command `cat`, which is described in Section 1, which is in *A/UX Command Reference*.

As a guide to the location of sections, you can refer to the general table of contents of each of the primary reference manuals, or to ''Current Organization of Sections into Books'' later in this preface. (The binder spines are also labeled with the section numbers, and occasionally section subdivisions, that are in each binder.)

Note also that there are a number of subcategory designations that can follow the digit reference in (1), (2), (3), (4), and (5), such as (1N). Detailed explanations of these subcategory designations are provided later in this preface.

Previous organization of sections into books

You may be curious about the logic behind the numbering of sections. The derivation of this numbering is much clearer when you realize that originally there was only one reference manual, the *UNIX User Manual*. In fact the manual pages were once considered the primary UNIX documentation, and the other books were originally considered supplements.

In the early days, all the manual pages easily fit into one book, in sections numbered 1 through 8. Section 8 originally contained the manual pages that are now located in Section 1M.

With the expansion of the original sections as UNIX grew, it became necessary to split the original book into several books, and this was done according to the audience they served. However, the original section numbering was preserved after the split because by then each number had come to have a particular meaning to UNIX users.

Because the original section numbers were preserved and then sections were recollated in accordance with the audience they served, the resulting books do not, for the most part, contain sequentially numbered sections.

The next section explains in detail how the sections are currently mapped into books.

There was another factor that led to the need to preserve the original section numbers. Some routines, system calls, and commands have the same names. To allow you to distinguish one from another, the section number is often included along with the name. While new section numbers could have helped distinguish these entities, the old numbers were much more familiar to UNIX users.

Besides distinguishing amongst identically named A/UX provisions, the section number helps identify each manual page as one that describes a command, a system call, a library routine, and so forth. Regular UNIX users sooner or later memorize what category is identified by each section number. After doing so, you can deduce how the sections must be split up into books—since each book serves a particular audience and each section category also goes along with a particular audience, the match-ups become fairly easy for you to make. The memorization part of this task is more or less considered an initiation rite for those who wish to learn to use UNIX effectively.

Until the 3.0 release of A/UX, the organization of sections into books was static. With the 3.0 release however, Section 7 has been moved out of *A/UX System Administrator's Reference* and into *A/UX Programmer's Reference*. This means that command provisions are now the exclusive focus of both *A/UX Command Reference* and *A/UX System Administrator's Reference*.

Current organization of sections into books

All manual pages are grouped by section. The sections are grouped by general function and are numbered according to standard conventions as follows:

1 User Commands

1M System Maintenance Commands

2 System Calls

3 Subroutines

4 File Formats

5 Miscellaneous Facilities

6 Games

7 Drivers and Interfaces for Devices

8 A/UX Startup Shell Commands

Each group or section of manual pages is located in one of the reference books. Each reference book may comprise more than one binder. This section explains where these sections are currently located with respect to the three primary reference books. It also describes any subcategories that may be present in a given section.

A/UX Command Reference contains Sections 1 and 6.

- Section 1—User Commands
  This section describes commands that require no special access privileges. The commands in Section 1 may also belong to a special category, such as networking commands. Where applicable, these categories are indicated by a letter designation that follows the section number. For example, the "N" in `ypcat`(1N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  1C Communications commands, such as `cu` and `tip`.

  1G Graphics commands, such as `graph` and `tplot`.

  1N Networking commands, such as those that help support various networking subsystems, including the Network File System (NFS), Remote Process Control (RPC) subsystem, and Internet subsystem.

- Section 6—Games
  This section contains all of the games provided with A/UX, such as `cribbage` and `worms`.

A/UX Programmer's Reference contains Sections 2 through 5 and Section 7.

- Section 2—System Calls
  This section describes the services provided by the A/UX system kernel, including the C language interface. It includes two special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "N" in `connect`(2N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  2N Networking system calls

  2P POSIX system calls

- Section 3—Subroutines
  This section describes the available subroutines. The binary versions of these subroutines are in the system libraries in the `/lib` and `/usr/lib` directories. The section includes seven special categories. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "N" in `mount`(3N) indicates that this manual page describes a networking command. Here is an explanation of each subcategory:

  3C C and assembly-language library routines

  3F Fortran library routines

  3M Mathematical library routines

  3N Networking routines

  2P POSIX routines

  3S Standard I/O library routines

  3X Miscellaneous routines

- Section 4—File Formats
  This section describes the structure of some files, but does not include files that are used by only one command (such as the assembler's intermediate files). The C language `struct` declarations corresponding to these formats are in the `/usr/include` and `/usr/include/sys` directories. There is one special category in this section, indicated by the letter designation "N" following the section number:

4N Networking formats

- Section 5—Miscellaneous Facilities
  This section describes various character sets, macro packages, and other miscellaneous facilities. There are two special categories in this section. Where applicable, these categories are indicated by the letter designation that follows the section number. For example, the "P" in `tcp`(1P) indicates a protocol. Here is an explanation of each subcategory:

  5F Protocol families

  5P Protocol descriptions

- Section 7—Drivers and Interfaces for Devices
  This section describes the drivers and interfaces through which devices are normally accessed. Access to one or more disk devices is fairly transparent when you are working with them in terms of files. When you want to use A/UX commands to communicate with devices more directly, at a level beyond the moderation of file systems, device files serve your needs. Such a level of communication permits you to request more explicit operating modes that may be supported by a disk (such as accessing disk partition maps), or that may be supported by other types of devices, such as tape drives and modems. For example, you can access a tape device in automatic-rewind mode as described in `tc`(7).

A/UX System Administrator's Reference contains Sections 1M and 8.

- Section 1M—System Maintenance Commands
  This section describes system maintenance programs such as `fsck` and `mkfs`.

- Section 8—A/UX Startup Shell Commands
  This section describes the commands that are available from within the A/UX Startup shell. This section includes detailed descriptions of the commands that contribute to the boot process and those that help with the maintenance of inactive file systems.

For more information

To find out where you need to go for more information about how to use A/UX, see *Road Map to A/UX*. This guide contains descriptions of each A/UX guide and ordering information for all the guides in the A/UX documentation suite.

Table of Contents

Section 1: User Commands (A-F)

NAME

intro — introduces the command and application programs

DESCRIPTION

This section describes, in alphabetical order, generally available commands. Certain distinctions of purpose are made using parenthetical designations in the guide words at the top of each page:

1C Specifies commands for communication with other systems.

1G Specifies commands used primarily for graphics and computer-aided design.

1N Specifies network commands.

DIAGNOSTICS

Upon termination, each command returns two bytes of status, one supplied by the system and giving the cause for termination, and (in the case of "normal" termination) one supplied by the program (see wait(2) and exit(2)). The former byte is 0 for normal termination; the latter is customarily 0 for successful execution and nonzero to indicate troubles such as erroneous parameters, bad or inaccessible data, or other inability to cope with the task at hand. It is called variously "exit code," "exit status," or "return code," and is described only where unusual conventions are involved. If present, this information is offered within the section entitled "Status Messages and Values."

WARNINGS

Some commands produce unexpected results when processing files containing null characters. These commands often treat text input lines as strings and therefore become confused upon encountering a null character (the string terminator) within a line.

NAME

300, 300s — filter text containing printer control sequences for a DASI terminal

SYNOPSIS

300 [+12] [-*half-line-units*] [-d*tab-delay*, *line-delay*, *char-delay*]

300s [+12] [-*half-line-units*] [-d*tab-delay*, *line-delay*, *char-delay*]

ARGUMENTS

+12

Permits use of 12-pitch, 6-lines-per-inch text. DASI 300 terminals normally allow only two combinations: 10-pitch, 6 lines per inch or 12-pitch, 8 lines per inch. To obtain the 12-pitch, 6-lines-per-inch combination, you should turn the PITCH switch to 12 and use the +12 option.

-d*tab-delay*, *line-delay*, *char-delay*

Specifies delay values for tabs (*tab-delay*), long line length (*line-delay),* and long strings of nonblank, nonidentical characters (*char-delay*). DASI 300 and 300s terminals sometimes produce peculiar output when faced with too many tab characters, very long lines, or long strings of nonblank, nonidentical characters. The 300 and 300s commands use delay values to adjust the timing of the output in these cases. Because terminal behavior varies according to the specific characters printed and the load on a system, you may need to override the default delay values, which are 3, 90, and 30, to get a satisfactory result. You can omit a value for *line-delay* and *char-delay*, or for just *char-delay*, to use their default delay values.

The commands insert one null (delay) character in a line for every set of tabs specified by *tab-delay* and for every contiguous string of nonblank, nontab characters specified by *line-delay*. If a line is longer than the number of bytes specified by *line-delay*, the commands perform the following calculation to determine the number of nulls to insert at the end of that line:

nulls = 1+(*total-line-length*)/20

If *tab-delay* or *char-delay* has a value of 0, the commands use two null bytes per tab or character, respectively. An option of -d0,1 may be appropriate for printing a C program that has many levels of indentation, and an option of -d3,30,5 may be appropriate for printing files such as /etc/passwd.

Note that the values supplied with the -d option interact with the prevailing carriage return and line-feed delays. The stty(1) modes nl0 and cr2 or nl0 and cr3 are recommended for most uses.

-half-line-units

Specifies the size of half-line spaces, thus allowing for individual taste in the appearance of subscripts and superscripts. A half-line is, by default, equal to 4 vertical plot increments. Because each increment equals 1/48 of an inch, a 10-pitch line feed requires 8 increments, while a 12-pitch line feed requires only 6. For example, you can make `nroff` half-lines to act as quarter-lines by using `-2`. You can also obtain appropriate half-lines for the 12-pitch, 8-lines-per-inch combination by setting the PITCH switch to 12 and by using the option `-3` alone.

DESCRIPTION

`300` supports special functions and optimizes the use of the DASI 300 (GSI 300 or DTC 300) terminal; `300s` performs the same functions for the DASI 300s (GSI 300s or DTC 300s) terminal. The `300` and `300s` commands convert half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions. The commands also draw Greek letters and other special symbols and permit convenient use of 12-pitch text. The commands reduce printing time up to 70 percent and can be used to print equations neatly as in the following sequence:

`neqn` *file...* `| nroff | 300`

The `neqn` names of, and resulting output for, the Greek and special characters supported by these commands are shown in `greek`(5).

You can use these commands with the `nroff -s` option or `.rd` requests to halt printing temporarily so that you can insert paper manually or change fonts in the middle of a document. Instead of pressing the RETURN key in these cases, press the line-feed key to continue printing.

In many (but not all) cases, the following two command lines are equivalent:

```
nroff -T300 files
nroff files | 300
```

Similarly, in many (but not all) cases, the following two command lines are equivalent:

```
nroff -T300 -12 files
nroff files | 300 +12
```

Thus, you can often avoid using `300` and `300s` unless special delays or options are required; in a few cases, however, the additional movement optimization of these commands may produce better-aligned output.

WARNINGS

If your terminal has a PLOT switch, set it to the "on" position before using `300`.

LIMITATIONS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there.

If your output contains Greek or reverse line feeds, use a friction-feed platen instead of a forms tractor. Although the forms tractor is good enough for drafts, it has a tendency to slip when reversing direction. This slippage causes distortion of Greek characters and misalignment of the first line of text after one or more reverse line feeds.

FILES

`/usr/bin/300`
Executable file `/usr/bin/300s` Executable file

SEE ALSO

`450`(1), `eqn`(1), `mesg`(1), `nroff`(1), `stty`(1), `tabs`(1), `tbl`(1), `tplot`(1G)

`greek`(5) in *A/UX Programmer's Reference*

See 300(1)

NAME

4014 — filters text containing printer control sequences a page at a time

SYNOPSIS

4014 [-c*columns*] [-n] [-p*lines*[i] [l [-t] [*file*]

ARGUMENTS

-c*columns*
: Specifies the number of columns to display and waits after displaying the last column.

-n
: Starts printing at the current cursor position and never erases the screen.

-p*lines*[i] [l]
: Specifies the page length in terms of its length in inches or lines. You can follow *lines* with an i for inches or with an l for lines, which is the default.

-t
: Omits pauses between pages. This option is useful for directing output to a file.

DESCRIPTION

4014 is intended for use with a Tektronix 4014 terminal. The 4014 command arranges for 66 lines to fit on the screen, divides the screen into columns, and contributes an 8-space page offset in the single-column case, which is the default. Tabs, spaces, and backspaces are collected and plotted when necessary. Teletype Model 37 half- and reverse-line sequences are interpreted and plotted. At the end of each page, 4014 waits for a new line (empty line) from the keyboard before continuing to the next page. In this wait state, the command !*cmd* sends *cmd* to the shell.

FILES

/usr/bin/4014
: Executable file

SEE ALSO

pr(1), tc(1), troff(1)

NAME

450 — filters text containing printer control sequences for the DASI terminal

SYNOPSIS

450

DESCRIPTION

450 supports special functions of, and optimizes the use of, the DASI 450 terminal or any terminal that is functionally identical, such as the Diablo 1620 or Xerox 1700. The 450 command converts half-line forward, half-line reverse, and full-line reverse motions to the correct vertical motions and draws Greek letters and other special symbols in the same manner as 300(1). You can use 450 to print equations neatly, as in this sequence:

neqn *file* ... | nroff | 450

The neqn names of, and resulting output for, the Greek and special characters supported by 450 are shown in greek(5).

You can use 450 with the nroff -s option or .rd requests to halt printing temporarily so that you can insert paper manually or change fonts in the middle of a document. Instead of pressing the RETURN key in these cases, press the line-feed key to continue printing.

In many (but not all) cases, the use of 450 can be eliminated in favor of one of the following commands:

nroff -T450*file...*

nroff -T450 -12*file...*

Thus, you can often avoid using 450 unless special delays or options are required; in a few cases, however, the additional movement optimization of 450 may produce better-aligned output.

WARNINGS

Make sure that the PLOT switch on your terminal is set to ''on'' before you use 450. You should set the SPACING switch to 10-pitch or 12-pitch. In either case, vertical spacing is 6 lines per inch unless you dynamically change the vertical spacing to 8 lines per inch by an appropriate escape sequence.

LIMITATIONS

Some special characters cannot be correctly printed in column 1 because the print head cannot be moved to the left from there. If your output contains Greek or reverse line feeds, use a friction-feed platen instead of a forms tractor. Although the forms tractor is good enough for drafts, it has a tendency to slip when reversing direction. This slippage causes distortion of Greek characters and misalignment of the first line of text after one or

more reverse line feeds.

FILES

`/usr/bin/450`
Executable file

SEE ALSO

`300`(1), `eqn`(1), `mesg`(1), `nroff`(1), `stty`(1), `tabs`(1), `tbl`(1), `tplot`(1G)

`greek`(5) in *A/UX Programmer's Reference*

NAME

adb — debugs executable programs

SYNOPSIS

adb [-k] [-w] [*object-file* [*core-file*]]

ARGUMENTS

core-file
: Specifies the name of a core image file that was produced when a core dump occurred while the object file specified by *object-file* was executing. The default value of *core-file* is core.

-k
: Causes adb to skip execution of a system call to gather relocation addresses. This option is useful when you are running adb on a stand-alone program, such as the kernel, (/unix), that does not have relocated addresses.

object-file
: Specifies the name of an executable program, preferably containing a symbol table. If the symbol table is not available, the symbolic features of adb cannot be used, although you can still use adb to examine the file. The default value of *object-file* is a.out.

-w
: Causes adb to open the object and core files for writing as well as reading. If the object file does not exist, adb creates it. You should use this option if you want to use adb to modify the object or core files. If you do not specify this option, adb opens the files for reading only.

DESCRIPTION

adb is a general-purpose debugger. You can use adb to examine core files and to debug object files in a controlled environment.

In general, requests to adb are of the form

[*address*]
[, *count*]
[*command*]
[;]

where *address* and *count* are expressions. If *address* is present, the current address, which is represented by a period (.) and is called ''dot,'' is set to *address*. Initially, dot is set to 0. For most commands, *count* specifies how many times the command is to be executed. The default value of *count* is 1.

The interpretation of an address depends on the context in which it is used. If a subprocess is being debugged, addresses are interpreted in the usual way in the address space of the subprocess. If the operating system is being debugged either post-mortem or by use of the special file

/dev/kmem to examine interactively or to modify memory, the maps are set to map the kernel virtual addresses. For further details of address mapping, see ''Addresses'' later in the ''Description'' section.

To quit adb, use the $q or the $Q command; see ''Commands'' later in the ''Description'' section. You can also press CONTROL-D.

The adb command ignores SIGQUIT, and SIGINT causes adb to return to the next adb command.

Expressions

You can form an expression from the following elements:

The last address typed.

'cccc'
The ASCII value of up to four characters. You can use a backslash (\ ) to escape a single quote (').

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

\+ The value of dot, incremented by the current increment.

. The value of dot.

<name
The value of *name*, which is either a variable name or a register name. The adb command maintains a number of variables (see ''Variables'' later in the ''Description '' section) named by single letters or digits. If *name* is a register name, the value of the register is obtained from the system header in *core-file*. The register names are those printed by the $r command.

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

_ symbol
In C, the actual name of an external symbol begins with an underscore. You may have to use the actual name to distinguish it from internal or hidden variables of a program.

integer
A number. The prefix 0 (zero) forces interpretation in octal radix; the prefixes 0d and 0D force interpretation in decimal radix; the prefixes 0x and 0X force interpretation in hexadecimal radix. Thus 020=0d16=0x10=16. If a prefix is not present, adb uses the default radix (see ''Commands'' later in the ''Description'' section for information on the $d command). The default radix is initially hexadecimal. The hexadecimal digits are 0123456789abcdefABCDEF. Note that a hexadecimal number whose most significant digit would otherwise be an alphabetical character

must have a `0x` (or `0X`) prefix (or a leading 0, if the default radix is hexadecimal).

integer`.`*fraction*
A 32-bit floating-point number.

symbol
Sequence of uppercase or lowercase letters, underscores, or digits, not starting with a digit. You can use a backslash (\ ) to escape other characters. The value of *symbol* is taken from the symbol table in the object file. An initial underscore (_) or tilde (~) is prefixed to *symbol*, if needed.

Monadic Operators

You can use these monadic operators with an expression:

`#`*exp*
Logical negation.

`*`*exp*
The contents of the location addressed by *exp* in *core-file.*

`-`*exp*
Integer negation.

`@`*exp*The contents of the location addressed by *exp* in *object-file.*

`~`*exp*
Bitwise complement.

Dyadic Operators

You can use these dyadic operators with two expressions. Dyadic operators are left-associative and are less binding than monadic operators.

e1`+`*e2*
Integer addition.

e1`-`*e2*
Integer subtraction.

e1`*`*e2*
Integer multiplication.

e1`%`*e2*
Integer division.

e1`&`*e2*
Bitwise conjunction.

e1`|`*e2*
Bitwise disjunction.

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

Commands

Most commands consist of a verb followed by a modifier or list of modifiers. The question mark (`?`) and slash (`/`) commands may be followed by an asterisk (`*`). See ''Addresses'' later in the ''Description'' section for further details. The following verbs are available:

newline
Repeats the previous command with a *count* value of 1 where *newline* is the ASCII character `0a`. By default, pressing the RETURN key produces a newline character.

`!` Calls a shell to read the rest of the line following `!`.

`>` *name*
Assigns dot to the variable or register named. This command is often used in the form *constant>name*. This form of the command can be used to enter 96-bit IEEE extended-precision numbers into the floating-point data registers `fp0-fp7`. For example, the following command puts the value 1.0 info `fp0`:

```
0x3FFF00008000000000000000 > fp0
```

When this form of the command is used, only the first 32 bits of the constant are stored in dot. See *MC68881 Floating Point Coprocessor User's Manual* (available from Motorola Literature Distribution Center, part number MC68881UM/AD), section 2.4, ''Extended Real,'' p. 211, for a description of IEEE extended-precision format.

`[?/]l` *value mask*
Masks words starting at dot with *mask* and compares them with *value* until a match is found. If `L` is used, the match is for 4 bytes at a time instead of 2. If no match is found, dot is unchanged; otherwise, dot is set to the matched location. If *mask* is omitted, -1 is used.

`[?/]m` *b1 e1 f1* `[?/]`
Records new values for *b1*, *e1*, *f1*. If fewer than three expressions are given, the remaining map parameters are left unchanged. If more than three expressions are given, the values of (*b2*, *e2*, *f2*), (*b3*, *e3*, *f3*) and so on, are changed. If the question mark (`?`) or slash (`/`) is followed by an asterisk (`*`), the first segment (*b1*, *e1*, *f1*) of the mapping is skipped, and the second and subsequent segments are changed instead. (There are as many (*bn*, *en*, *fn*) triples as you have sections in your program.) If the list is terminated by `?` or `/`, *object-file* or *core-file*, respectively, is used for subsequent requests. For example, `/m?` causes `/` to refer to the object file.

[?/]w *value* ...
Writes the 2-byte value *value* into the addressed location. If the command is W, adb writes 4 bytes. Odd addresses are not allowed when you are writing to the subprocess address space.

You can place a format request after the /, =, and ? commands to specify a style of printing:

/*format*
Prints locations starting at *address* in *core-file* according to *format*, and dot is incremented as for the question mark.

=*format*
Prints the value of *address* itself in the styles indicated by *format*. (For i format, a question mark is printed for the parts of the instruction that reference subsequent words.)

?*format*
Prints the locations starting at *address* in *object-file* according to *format*. Dot is incremented by the sum of the increments for each format letter.

A format consists of one or more characters. Each format character may be preceded by a decimal integer that is a repeat count for the format character. As the format is stepped through, dot is incremented by the amount given for each format letter. If no format is given, the last format is used. These format letters are available:

"..." 0
Prints the enclosed string.

\+ Increments dot by 1. Nothing is printed.

\- Decrements dot by 1. Nothing is printed.

^ Decrements dot by the current increment. Nothing is printed.

a 0
Prints the value of dot in symbolic form. Symbols are checked to ensure that they have an appropriate type, where / is a global data symbol, ? is a global text symbol, and = is a global absolute symbol.

b 1
Prints the addressed byte in octal.

C 1
Prints the addressed character according to the standard escape convention, where control characters are printed as ^X and the delete character is printed as ^?.

c 1
Prints the addressed character.

`D` 4
Prints as a long decimal number.

`d` 2
Prints as a decimal number.

`F` 8
Prints as a double floating-point number.

`f` 4
Prints the 32-bit value as a floating-point number.

`i` *n*
Disassembles the addressed instruction.

`n` 0
Prints a new line.

`O` 4
Prints 4 bytes as an octal number.

`o` 2
Prints 2 bytes in hexadecimal. All octal numbers output by `adb` are preceded by 0.

`p` 4
Prints the addressed value in symbolic form, using the same rules for symbol lookup as `a`.

`Q` 4
Prints as a long signed octal number.

`q` 2
Prints as a signed octal number.

`r` 0
Prints a space.

`S` *n*
Prints a string using the ^*X* escape convention (see `C` earlier in this list); *n* is the length of the string, including its 0 terminator.

`s` *n*
Prints the addressed characters until a 0 character is reached.

`t` 0
Tabs to the next appropriate tab stop when preceded by an integer. For example, `8t` causes a move to the next 8-space tab stop.

`U` 4
Prints as a long unsigned decimal number.

`u` 2
Prints as an unsigned decimal number.

`Y` 4
Prints 4 bytes in data format (see `ctime`(3)).

`X` 4
Prints 4 bytes as a hexadecimal number.

`x` 2
Prints 2 bytes as a hexadecimal number.

$*modifier*
These are miscellaneous commands. These modifiers are available:

<*file*
Reads commands from *file*. If this command is executed in a file, further commands in the file are not seen. If *file* is omitted, the current input stream is terminated. If the value of *count* is 0, the command is ignored. The value of the count is placed in variable `9` before the first command in *file* is executed.

<<*file*
Reads commands from *file*. This command is similar to < except it can be used in a file of commands without causing the file to be closed. Variable `9` is saved during the execution of this command and restored when the command completes. There is a (small) finite limit to the number of << files that can be open at once.

>*file*
Appends output to *file*, which is created if it does not exist. If *file* is omitted, output is returned to the terminal.

`?` Prints process ID, the signal that caused stoppage or termination, as well as the registers in the same way as the `$r` command. This option is the default if *modifier* is omitted.

`b` Prints all breakpoints and their associated counts and commands.

`c` Performs a C stack backtrace. If *address* is given, it is taken as the address of the current frame (instead of `a7`). If `C` is used, the names and (16-bit) values of all automatic and static variables are printed for each active function. If *count* is given, only the first *count* frames are printed.

`d` Sets the default radix to *address* and reports the new value. Note that *address* is interpreted in the (old) current radix. Thus *10$d* never changes the default radix. To make the default radix decimal, use *0t10$d*.

`d` Resets integer input as described in ''Expressions'' earlier in the ''Description'' section.

`e` Prints the names and values of external variables.

`f` Prints the floating-point data registers *fp0-fp7* in IEEE extended precision (see *>name*, earlier in the ''Commands'' section for a definition), and exponential notation, along with the floating-point control registers `fpcr`, `fpsr`, and `fpiar`.

`m` Prints the address map.

`o` Regards all integers subsequently input as octal.

`Q` Exits from `adb`.

`q` Exits from `adb`.

`r` Prints the general registers and the instruction addressed by `pc`. Dot is set to `pc`.

`s` Sets the limit for symbol matches to *address*. The default is 255.

`v` Prints all nonzero variables in hexadecimal.

`w` Sets the page width for output to *address*. The default is 80.

:*modifier*

The colon (`:`) command and a modifier are used to manage a subprocess. These modifiers are available:

`bc` Sets breakpoint at *address*. The breakpoint is executed *count* - 1 times before causing a stop. Each time the breakpoint is encountered, the command *c* is executed. If this command is omitted or sets dot to 0, the breakpoint causes a stop.

`cs` Continues the subprocess with signal *s* `cs` (see `signal`(3)). If *address* is given, the subprocess is continued at this address. If no signal is specified, the signal that caused the subprocess to stop is sent. Breakpoint skipping is the same as for the `:r` command.

`d` Deletes breakpoint at *address*.

`k` Terminates the current subprocess, if any.

`r` Runs *object-file* as a subprocess. If *address* is given explicitly, the program is entered at this point; otherwise, the program is entered at its standard entry point. The value of *count* specifies how many breakpoints are to be ignored before stopping. Arguments to the subprocess can be supplied on the same line as the command. An argument starting with < or > causes the standard input or output to be established for the command. All signals are turned on upon entry to the subprocess.

`ss` Continues the subprocess as described for `c`, earlier in this list, except that the subprocess is single-stepped the number of times specified by *count*. If there is no current subprocess, the object file is run as a subprocess in the same way as for the `:r` command. In this case, no signal can be sent; the remainder of the line is treated as arguments to the subprocess.

Variables

The `adb` command provides a number of variables. Named variables are set initially by `adb` but are not used subsequently. These numbered variables are reserved for communication:

`0` The last value printed.

`1` The last offset part of an instruction source.

`2` The previous value of variable 1.

`9` The count on the last `$<` or `$<<` command.

On entry, the following variables are set from the system header in *core-file*. If *core-file* does not appear to be a core file, these values are set from *object-file*:

`b` The base address of the data segment.

`d` The data segment size.

`e` The entry point.

`m` The magic number (0407, 0410 , 0413).

`s` The stack segment size.

`t` The text segment size.

Addresses

The address in a file associated with a written address is determined by a mapping associated with that file. Each mapping is represented by *n* triples (*b1*,*e1*,*f1*), (*b2*,*e2*,*f2*), ... (*bn*,*en*,*fn*), corresponding to the number of sections in your object file. The file address corresponding to a written address is calculated as follows:

b1≤*address*<*e1*
⇒
file-address=*address*+*f1*-*b1*

otherwise,

b2≤*address*<*e2*
⇒
file-address=*address*+*f2*-*b2*

and so on.

Otherwise, the requested address is not legal. In some cases (such as for programs with separated instruction and data space), the two segments for a file can overlap. If a question mark (`?`) or slash (`/`) is followed by an asterisk (`*`), the first triple is not used.

The initial setting of both mappings is suitable for normal `a.out` and `core` files. If either file is not of the kind expected, for that file *b1* is set to 0, *e1* is set to the maximum file size, and *f1* is set to 0; in this way the whole file can be examined with no address translation.

So that `adb` can be used on large files, all appropriate values are kept as signed 32-bit integers.

EXAMPLES

This command starts `adb` on the executable file called `obj1`:

```
adb obj1
```

When `adb` responds with a `ready` message, you can type this request:

```
main,10?ia
```

The request causes 16 (10 hexadecimal) instructions to be printed in assembly language, starting from `main`.

STATUS MESSAGES AND VALUES

The `adb` command echoes `adb` when there is no current command or format.

The `adb` command produces comments about, for example, inaccessible files, syntax errors, and abnormal termination of commands.

The exit status of `adb` is 0 unless the last command failed or returned a nonzero status.

LIMITATIONS

Use of the number sign (`#`) for the unary logical negation operator is peculiar.

There doesn't seem to be any way to clear all breakpoints.

In certain cases, disassembled code cannot be used directly as input to `as`(1). This is because `adb` gives more useful information than `as` accepts. For example, explicit register names are given in the disassembly of `movm` and `fmovm` instructions.

FILES

`/bin/adb`
: Executable file

`a.out`
: Default object file

`core`
Default core file

SEE ALSO

`sdb`(1)

`a.out`(4), `core`(4) in *A/UX Programmer's Reference*

NAME

addbib — creates or extends a bibliographic database

SYNOPSIS

addbib [-a] [-p *prompt-file*] *database*

ARGUMENTS

-a Suppresses the default prompting for an abstract.

database

Specifies the name of a file to be used to store the output of addbib. If the file specified by *database* does not exist, addbib creates it. If the file already exists, addbib appends to it any entries you made.

-p *prompt-file*

Causes addbib to use prompts that are defined in *prompt-file*. This file should contain lines that consist of a prompt string, a tab, a percent sign (%), and the option, in that order.

DESCRIPTION

addbib creates or extends a bibliographic database. The structure of the database allows formatting to be imposed as a separate step after the data is entered. Database entries consist of options and relevant fields. Once you have entered the data, you can use sortbib(1) and roffbib(1) to print the database in a standard bibliographic format. You can also embed keywords from the database in footnotes in nroff(1) or troff(1) documents and use refer(1) to extract the complete reference from the database in proper footnote format and print it in your document.

When started, addbib displays the Instructions? prompt. Entering y causes addbib to print a summary of how to enter data. You can enter n or press RETURN to skip the summary.

Next, addbib prompts for various bibliographic fields and reads the response from the terminal. The addbib command does not actually write the fields to the database until all the fields for one record have been prompted. If you have no data for a particular field, press RETURN to go on to the next prompt.

To enter data, type the information and press RETURN. The only exception to this practice occurs when you enter data in response to the Abstract: prompt. In this case, type the data, press RETURN, and then press CONTROL-D. If you wish to enter no data in response to the Abstract: prompt, press CONTROL-D. To continue any field on the next line, enter a backslash (\). In response, addbib places a > prompt on the next line, where you can enter more data. When the field is written to the database, the second and any additional continuation lines are separated by the newline character.

You can enter a minus sign (-) to go back to a previous prompt and add a second field of a particular type. For example, you can use this feature to enter the name of each author of a multiple-author book. You cannot use this feature to overwrite a previously entered field.

When addbib displays the Continue? prompt, which appears after you have entered one complete record, you can enter y or press RETURN to continue or enter n to stop running addbib. You can also enter the name of a text editor (vi, ex, edit, or ed) to edit the database.

The addbib command insulates you from the options by displaying an equivalent "English" prompt for each option. By default, addbib displays prompts for the A, T, J, V, P, I, C, D, O, K, and X options. Here are the common options and their meanings:

%A Author's name.

%B Book containing article referenced.

%C City (place of publication).

%D Date of publication.

%E Editor of book containing article referenced.

%F Footnote number or label (supplied by refer).

%G Government order number.

%H Header commentary, printed before reference.

%I Issuer (publisher).

%J Journal containing article.

%K Keywords to use in locating reference.

%L Label field used by -k option of refer.

%N Number within volume.

%O Other commentary, printed at end of reference.

%P Page number(s).

%Q Corporate or foreign author (unreversed).

%R Report, paper, or thesis (unpublished).

%S Series title.

%T Title of article or book.

%V Volume number.

%X Abstract; used by roffbib, not by refer.

`%Y` Ignored by `refer`.

`%Z` Ignored by `refer`.

Except for `A`, each field should be given just once.

EXAMPLES

Here is a record of the data entry for one bibliographic reference using the default prompts:

```
Instructions? n

   Author:  R. Pike
    Title:  -
   Author:  B. W. Kernighan
    Title:  Program Design in the UNIX System Environment
  Journal:  Technical Journal
   Volume:  63 No. 8 Part 2
    Pages:  1595-1605
Publisher:  AT&T Bell Laboratories
     City:  Short Hills, NJ
     Date:  October 1984
    Other:
 Keywords:  Programming UNIX
 Abstract:  (ctrl-d to end)

Continue? n
```

Here is what the database contains:

```
%A R. Pike
%A B. W. Kernighan
%T Program Design in the UNIX System Environment
%J Technical Journal
%V 63 No. 8 Part 2
%P 1595-1605
%I AT&T Bell Laboratories
%C Short Hills, NJ
%D October 1984
%K Programming UNIX
```

LIMITATIONS

Because `addbib` displays only the first 20 characters, the prompt strings in a user-defined prompt file should be less than or equal to 20 characters. If the prompt string is longer than 20 characters, `addbib` appends the option from the prompt file to the end of the truncated prompt string.

FILES

`/usr/ucb/addbib`
Executable file

SEE ALSO

`indxbib`(1), `lookbib`(1), `refer`(1), `roffbib`(1), `sortbib`(1)

NAME

admin — creates and administers SCCS files

SYNOPSIS

admin [-a*name-or-gid*] [-d*option*[*value*]] [-e*name-or-gid*] [-f*option*[*value*]] [-h] [-i[*name*]] [-m[*mrlist*]] [-n] [-r*release*[.*level*]] [-t[*descriptive-text*]] [-y[*comment*]] [-z] *file...*

ARGUMENTS

-a*name-or-gid*

Specifies a login name or a numeric group ID to be added to the list of users who are allowed to modify the Source Code Control System (SCCS) file. The list of users is stored in the control information of the SCCS file. Specifying a group ID is the same as specifying all the login names common to that group ID. You can use more than one -a option on a single admin command line, and you can accumulate as many login names and numeric group IDs as you wish. If the control information does not include a list of users, which is the default, anyone who can read and write the file can check out the file for editing. If you precede *name-or-gid* with an exclamation mark (!), the specified login name or members of the specified group ID will not be allowed to check out the file for editing. To remove a login name or group ID from the list of users, see the -e option.

-d*option* [*list*]

Causes the specified option, previously added by the -f option, to be removed from the SCCS file. You can specify the -d option on existing SCCS files only, and you can combine several -d options on a single admin command line. If you specify the l option and a list of releases, admin "unlocks" the releases. See the -f option for the other possible options.

-e*name-or-gid*

Specifies a login name or numeric group ID to be removed from the list of users who are allowed to make changes to the file. The list of users is stored in the control information of the SCCS file. Specifying a group ID is the same as specifying all login names common to that group ID. You can use several -e options on a single admin command line. This option undoes the work of the -a option.

file Specifies the file that admin is to create or update. When a file is placed under SCCS for the first time, *file* is the name of a file that admin is to create. The number of *file* arguments depends on the option, that you use to create the file. (These options are described later in this list. For compatibility with the SCCS commands, the name of a file controlled by SCCS must begin with an s and a period (.). The name that you provide can consist of up to 255 characters,

including the required s. prefix. When creating a new file, admin initializes control information according to the options provided on the command line and assigns default values for any required options that are not specified.

When changing control information, you can provide one or more *file* arguments, which is the name of a file previously created by the admin command. The admin command changes only the specified control information and leaves the remainder alone. If *file* is a directory, admin behaves as though each file in the directory were specified on the command line. If *file* is a hyphen (-), admin reads the standard input and interprets each line as the name of an SCCS file to be processed. In either case, files that do not begin with s. and unreadable files are silently ignored.

-f*options* [*list*]

Specifies an option to be placed in the control information of an SCCS file. Some options require or accept *value*, which further defines the action of *option*. You can use more than one -f option on a single admin command line. You can use the -d option to remove an option from an SCCS file. Here are the allowable values of *option* and their meanings:

b Allows the use of the -b option on any future get command to create branch deltas.

c*ceiling*

Specifies the highest release number that get can assign to this file. The value of *ceiling* must be less than 10,000. The default value of *ceiling* is 9999. See the -r option for an explanation of release numbers.

d*deltanum*

Specifies the default delta number (SID) to be used by any future get command.

f*floor*

Specifies the lowest release number that get can assign to this file. The value of *floor* must be greater than 0 but less than 9999. The default value of *floor* is 1. See the -r option for an explanation of release numbers.

i[*string*]
Protects the presence of keywords, which you add manually to the text of a file before you run `admin` on it. If a future `get` or `delta` of a protected file results in the `No id keywords (ge6)` message, the message is treated as a fatal error. You must restore the keywords to the file before proceeding. If you do not use this option, `get` and `delta` treat this message as a warning.

If you do not specify *string*, all keywords are protected. If you specify *string*, the SCCS file must contain a string of keywords that exactly matches *string*. Embedded newlines are not allowed in *string*. To specify multiple keywords separated by a space, quote them, as in

```
-fi"%M %I %D"
```

See `get`(1) for a list of valid identification keywords.

j Allows concurrent `get` commands for editing the same SID of an SCCS file. This option allows multiple concurrent updates to the same version of the SCCS file.

l*list*
Specifies a list of releases to which deltas may no longer be made. Any future attempts to use the `get` command on one of these ''locked'' releases for editing will fail. The list has the following syntax:

<list> `::=` *<range>* | *<list>* , *<range>*
<range>~ `::=` *release-number* | a

Using the character `a` in *list* is equivalent to specifying all releases for the named SCCS file.

m*module-name*
Uses *module-name* to specify a string that `get` will substitute for all occurrences of the `%M%` keyword in the SCCS file when the text is retrieved. If you do not use the `m` option, `get` substitutes the name of the SCCS file, with the prefix `s.` removed, for all occurrences of the `%M%` keyword.

n Causes future invocations of `delta` to create a null delta for any skipped releases when a delta is made in a new release. For example, if you make delta 5.1 after delta 2.7, releases 3 and 4 are skipped. The null deltas serve as anchor points so that you can create branch deltas later. If you do not use this option, you will be unable to create branch deltas for the skipped releases in the future.

q*string*
Specifies a string that `get` will substitute for all occurrences of the `%Q%` keyword in the SCCS file when the text is retrieved.

t*module-type*
Uses *module-type* to specify a string that `get` will substitute for all occurrences of the `%Y%` keyword in the SCCS file when it retrieves the text.

v[*program*]
Uses *program* to specify the name of a user-supplied Modification Request (MR) validity-checking program. The value of *program* can be an absolute pathname or a program that resides in a directory listed in `$PATH`. If you set this option when you are creating an SCCS file, you must also use the `-m` option.

The `v` option causes future invocations of `delta` on the text of the SCCS file to prompt for MR numbers as the reason for making a delta and to invoke the validity-checking program to verify that the MR number is valid.

`-h` Causes `admin` to check the structure of the SCCS file (see `sccsfile`(4)) and to compare a newly computed checksum with the checksum stored in the first line of the SCCS file. (The checksum is the sum of all the characters in the SCCS file except those in the first line.) The `admin` command displays an error diagnostic if the comparison of the checksums fails.

Because this option inhibits writing to the *file* argument, you cannot use this option in conjunction with any other option. As a result, this option is useful only when an existing file is processed.

`-i`[*name*]
Creates a new SCCS file whose text is the content of the file specified by *name*. If you use this option and omit *name*, `admin` obtains the text by reading the standard input until it encounters an end-of-file. You can specify only one *file* argument when you use this option. If you do not use this option, you must explicitly specify the `-n` option.

`-m`[*mrlist*]
Specifies a list of alphanumeric MR numbers to be associated with the initial version of an SCCS file. This option can be used only with the `-i` or `-n` option and must be used in conjunction with the `v` option.

If the list has more than one MR number, enclose the list in quotation marks and separate the numbers by a space or a tab. This list can have up to 61 MR numbers.

The admin command executes the user-supplied MR validity-checking program specified by the v option to verify *mrlist*. The admin command displays error diagnostics if the v option is not set or if MR validity checking fails. If validity-checking is successful, the admin command inserts *mrlist* in a manner identical to that of delta.

-n Creates a new SCCS file that contains only control information. Use this option when you do not have an existing file to place under SCCS. You can supply more than one *file* argument when you use this option.

-r*release*[.*level*]
Specifies an integer value for the release and level into which the initial delta is to be inserted. If you do not use this option, 1.1 is the default initial delta. You can use this option only if you also use the -i option.

-t[*descriptive-text*]
Causes descriptive text, from the file specified by *descriptive-text*, to be incorporated in the control information of the SCCS file. Descriptive text is any text that, for your own purposes, you want to associate with the actual text of the file.

If you use this option with the -i or -n option to create an SCCS file, *descriptive-text* is required. If you use this option with an existing SCCS file and do not provide *descriptive-text*, admin removes any existing descriptive text from the SCCS file. If you provide *descriptive-text*, admin replaces any existing descriptive text with the contents of the file specified by *descriptive-text*.

-y[*comment*]
Specifies text to be inserted as a comment in the SCCS file when it is created. If *comment* contains spaces, enclose the entire comment in double quotation marks. You can use this option only in conjunction with the -i and -n options. The admin command inserts the comment in a manner identical to that of delta. If you do not specify this option, admin inserts a default comment of the form:

date and time created *YY/MM/DD HH:MM:SS* by *login-id*

-z Causes admin to recompute the checksum and store it in the first line of the SCCS file. Note that use of this option on a corrupted file prevents future detection of the corruption.

DESCRIPTION

admin creates new SCCS files and changes the control information of existing SCCS files. A file created by admin contains, in addition to the actual text, control information used by the SCCS commands to manage

the text. The control information consists of an option followed by the associated information.

In general, you place a file under SCCS by using the `-i` or `-n` option, and you tailor the control information through the other options. Many of the options for tailoring the control information manage the outcome of future `get` and `delta` commands. Any particular version of an SCCS file is commonly called a ''delta''; the cycle of running `get` and `delta` on a file is commonly called ''checking a file out and in.'' Making a change to an SCCS file is commonly called ''making a delta.''

Security and Permission Bits

For the highest level of security, the permission bits of directories that contain SCCS files should be 755, and the permission bits of SCCS files themselves should be read-only (444). These settings allow only the owner of an SCCS file to modify it by using SCCS commands only.

To create a new SCCS file, you must have write permission in the pertinent directory. The `admin` command sets the permission bits on a newly created SCCS file to 444. When updating an existing SCCS file, `admin` retains the file's original permission bits. The `admin` command writes to a temporary file called `x.`*filename*. This file is created with read-only permission bits, if a new SCCS file is being created; if an existing file is being changed, this file is created with the same permission bits as the existing SCCS file. After successfully creating or updating `x.`*filename*, `admin` removes the SCCS file (if it exists) and renames `x.`*filename* with the name of the SCCS file. This strategy ensures that changes are made to the SCCS file only if no errors occurred.

The `admin` command also uses a temporary lock file, `z.`*filename*, to prevent simultaneous updates to an SCCS file by two or more users. See `get`(1) for further information.

Handling a Corrupted File

If you, as the owner, need to correct an SCCS file by using a non-SCCS command, you can change the permission bits to 644 and use an editor to make the correction. You should always run `admin -h` on the edited file to check for corruption and then run `admin -z` to generate a valid checksum. You should then run `admin -h` again to ensure that the SCCS file is valid.

The Validity-Checking Program

If you want to take advantage of the `v` option, which causes `delta` to prompt for MR numbers, you must write a program to check the numbers. The `admin` and `delta` commands call `execvp` (see `exec`(2) for details) to execute the validity-checking program. The first element of the `argv` array contains the name of the validity-checking program, as specified by

the v option, and the second element contains the filename, with its s. prefix removed, of the SCCS file being processed. The validity-checking program should return zero to indicate that the MR number is valid, or nonzero to indicate failure.

EXAMPLES

This command uses a file named tempest.c to create a new file in SCCS format named s.tempest:

```
admin -itempest.c s.tempest.c
```

STATUS MESSAGES AND VALUES

The bdiff command produces messages that the help command can interpret. You may, for example, see this message:

```
ERROR [s.file]: MRs required (de10)
```

To see an explanation of this message, enter

```
help de10
```

FILES

/usr/bin/admin
: Executable file

filename
: Existing file to be placed under SCCS control

s.*filename*
: SCCS file created or changed by admin

x.*filename*
: Temporary file

z.*filename*
: Lock file that prevents simultaneous updates to s.*filename*

SEE ALSO

delta(1), ed(1), get(1), help(1), prs(1), what(1)

sccsfile(4) in *A/UX Programmer's Reference*

"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*

NAME

apply — passes its arguments in batches to a command that is run once per every batch

SYNOPSIS

apply [-a*esc-char*] [*-args-per-batch*] *command argument...*

ARGUMENTS

-a*esc-char*
: Specifies the escape character that can change the interpretation of *argument*. The default is the percent sign (%). See the ''Description'' section for information about how the argument selector works.

argument
: Specifies an argument to be passed to *command*. You should specify a multiple of the number of arguments that are needed to run *command* successfully once.

command
: Specifies the command that apply is to run.

-args-per-batch
: Specifies the number of arguments to pass from the apply command line to *command* each time apply runs the command. If you do not use this option, apply passes one argument at a time. If the value of *args-per-batch* is 0, apply runs *command* once for each *arg* but does not pass *arg* to *command*.

DESCRIPTION

apply runs a command enough times to use up the arguments specified on the apply command line.

An argument-selector is part of a command and has the form %*d*, where *d* is a digit from 1 to 9. The apply command replaces the argument-selector with the next digit *argument* and runs the command. If an argument-selector is present in *command*, apply ignores the *-args-per-batch* option, if provided. To avoid an error message when the arguments are exhausted, you need to make the number of arguments on the apply command line a multiple of *d*. See the third example in the next section for a way to use an argument-selector.

EXAMPLES

The first example of the apply command uses cmp to compare the a files to the b files. The -2 option causes apply to take arguments in sets of two from the apply command line and use them to run cmp until the arguments are exhausted.

```
apply -2 cmp a1 b1 a2 b2
```

The second example shows what happens when the -*n* option is set to 0.

Five arguments appear after the who command, so apply runs the who command five times, but it does not pass the arguments to who because *n* is 0.

```
apply -0 who a 2 c 4 e
```

The third example shows the use of an argument-selector. This command creates a link in the directory /usr/mcfong between each file in the current directory:

```
apply 'ln %1 /usr/mcfong' *
```

If the argument-selector were %3, the apply command would run ln for every third filename expanded by the shell. If the total number of expanded filenames were not a multiple of 3, apply would run ln for every third filename and display an error message when it could not get the next argument that is a multiple of 3.

LIMITATIONS

Shell metacharacters in *command* may have bizarre effects. You should enclose complicated commands in single quotation marks (').

You cannot pass a literal %2 if % is the argument-selector character.

FILES

/usr/ucb/apply
Executable file

SEE ALSO

csh(1), ksh(1), sh(1), xargs(1)

NAME

apropos — locates commands by keyword

SYNOPSIS

apropos *search-string...*

ARGUMENTS

search-string

Specifies a string for which apropos is to search.

DESCRIPTION

apropos examines a database of manual page ''Name'' sections for the occurrence of the specified strings. If a match is found, apropos displays the command name and its corresponding ''Name'' section. You can use apropos to help you find a command that does the task you want done. The apropos command separately considers each *search-string*, ignoring case. A string that is part of another word is a match; thus, a search string of compile matches compiler.

EXAMPLES

This command searches for the string calendar:

```
apropos calendar
```

The command displays this output:

```
cal(1)      - generate a calendar for the specified year
calendar(1) - reminder service
```

The first column contains the names of commands whose ''Name'' section contains calendar, followed by the section number, in the form *name*(*section*). The second column contains the actual ''Name'' section.

To see the on-line documentation for a particular command, enter a command of this form:

```
man [section] name
```

For instance, enter this command:

```
man 1 cal
```

FILES

/usr/ucb/apropos

Executable file

/usr/lib/whatis

Database that apropos searches

SEE ALSO

man(1), whatis(1)

NAME

ar — maintains a library of files in an archive

SYNOPSIS

ar -dp [l] [v] *archive file...*

ar -mp [l] [v] [*position archivefile*] *archive file...*

ar -qp [c] [l] [v] *archive file...*

ar -rp [c] [l] [u] [v] [*position archivefile*] *archive file...*

ar -tp [s] [v] *archive file...*

ar -xp [l] [s] [v] *archive file...*

ARGUMENTS

archive
Specifies the name of the archive to be created or maintained.

-c Causes ar to create an archive without displaying a message. This option is useful only when used in combination with the -r and -q options.

-d Deletes the named files from the archive.

file Specifies the name of a file that is to be added to the archive or that is already in the archive.

-l Causes ar to place its temporary files in the current directory rather than in the default /tmp. This option is useful only when used in combination with the -d, -m, -q, -r, and -x options.

-m Moves the named files to the end of the archive by default. You can combine this option with a position specifier to move the file before or after a file that is already in the archive.

-p Prints the contents of files that are in the archive as specified by *file*.

position archivefile
Specifies the position specifiers, where *archivefile* is the name of a file in the archive. Replace *position* with one of these options:

-a Moves the named files after *archivefile*.

-b Moves the named files before *archivefile*.

-i Inserts the named files before *archivefile*. This action is identical to the action of the -b position specifier.

-q Appends the named files to the end of the archive. If the archive does not exist, ar creates it and displays the message ar: creating archive. You cannot use a position specifier, as described for the -m option, with this option. The -q option does not check whether the named file is already in the archive and therefore can add

duplicates. This option is used to avoid quadratic behavior when creating a large archive on a file-by-file basis. For an alternative to the `-q` option, see the `-r` option.

`-r` Replaces the named file in the archive. If the named file is not already in the archive, `ar` adds the file to end of the archive unless you also use a position specifier, as described for the `-m` option. If the archive does not exist, `ar` creates it and displays the message `ar: creating archive`.

`-s` Causes `ar` to regenerate the archive's symbol table. The `-d`, `-m`, `-q`, `-r`, and `-u` options cause `ar` to regenerate the symbol table automatically. The other options do not. You can use this option in conjunction with the `-p`, `-t`, and `-x` options to restore the symbol table, when, for example, `strip` has been used on the archive.

`-t` Prints the name of each named file in the archive. If you do not specify any names, `ar` prints the names of all files in the archive. Typically, you use this option with the `-v` option.

`-u` Changes the behavior of the `-r` option by causing `ar` to replace only those files whose modification dates are more recent than the modification date of the archive.

`-v` Causes `ar` to give a file-by-file description as it works. When used with the `-t` option, `ar` prints a listing similar to the output of the `ls`(1) command for each file in the archive.

`-x` Extracts a copy of the named file. If you do not specify a name, `ar` extracts all files in the named archive. In either case, this option does not alter the archive.

DESCRIPTION

`ar` maintains files in a single file that is called an ''archive.'' Typically, an archive is a library of object files that is used by the link editor, `ld`, to resolve references so that it can produce an executable file. You can also use `ar` to create and maintain libraries of other file types.

When `ar` creates an archive, headers are created in a format that is portable across all machines. The portable archive format and structure are described in detail in `ar`(4). The archive symbol table (described in `ar`(4)) is used by the link editor (`ld`) to do multiple passes over libraries of object files efficiently. Whenever you use `ar` to create or update an archive, `ar` rebuilds the symbol table. You also can use the `-s` option to rebuild the symbol table.

EXAMPLES

The first example replaces the file `foo.o` in the archive `libfoo.a` with a new copy of `foo.o`:

```
ar -rc libfoo.a foo.o
```

If `libfoo.a` does not exist, `ar` creates it. The `-c` option prevents `ar` from issuing a message that it has created the archive. Because the command does not use a position specifier, `ar` places `foo.o` at the end of the archive if `foo.o` is not already in the archive.

The second example uses the `-u` option to cause `ar` to update the archive using only those files in the current directory that have a `.o` suffix and that are newer than the modification date of the archive itself:

```
ar -ru libfoo.a *.o
```

The third example inserts the file `new.o` into the archive `libfoo.a` before `foo.o`, which is already in the archive:

```
ar -rvb foo.o libfoo.a new.o
```

The `-v` option causes `ar` to print a message as it adds `new.o`.

LIMITATIONS

If you specify the same file more than once in an argument list, `ar` puts the file in the archive once for each mention.

FILES

`/bin/ar`
: Executable file

`/tmp/ar*`
: Temporary files

SEE ALSO

`ld`(1), `lorder`(1), `strip`(1), `tar`(1)

`a.out`(4), `ar`(4) in *A/UX Programmer's Reference*

NAME

as — assembles files by translating assembler mnemonics to object code

SYNOPSIS

as [-A *factor*] [-m] [-n] [-o *object-file*] [-R] [-V] [-68030] [-68040] [-68851] *file*

ARGUMENTS

-A *factor*
Specifies the expansion factor to be used to increase the size of the default symbol table.

file Specifies the name of the file to be assembled. By convention, assembly-language filenames have the .s suffix. If you specify more than one *file* argument, as assembles only the last-named file.

-m Causes as to run the m4 macro preprocessor on *file* and assemble the output of m4.

-n Turns off address optimization. By default, as optimizes addresses by replacing, where possible, a reference to a long address with a reference to a short relative address.

-o *object-file*
Causes as to put its output in *object-file*. If you do not use this option, as puts its output in a file whose name is formed by removal of the .s suffix, if there is one, from *file* and substitution of the .o suffix.

-R Causes as to remove *file* when assembly is completed. By default, this option is off.

-V Causes as to write its version number on the standard error.

-68030
Assemble for the MC68030 processor and MC68030 MMU. This option give you access to an enhanced feature set as compared to the default MC68020 assembly, but the code does not run on the original Macintosh II computer.

-68040
Assemble for the MC68040 processor and MC68040 MMU. This option give you access to an enhanced feature set as compared to the default MC68020 assembly, but the code does not run on the original Macintosh II computer.

-68851
Assemble for the MC68851 Memory Management Unit (MMU). This command is on by default.

DESCRIPTION

as assembles assembly-language files. The C and Fortran compilers produce assembly-language files and automatically call as to assemble them.

WARNINGS

If you use the -m option, *file* cannot use the names of m4 built-in macros as names for variables, functions, or labels. This is because m4 cannot distinguish between the use of the built-in macro names as macros and as assembler symbols (see m4(1)).

LIMITATIONS

The as command cannot process arithmetic expressions that have more than one forward-referenced symbol per expression.

FILES

/bin/as
: Executable file

/usr/tmp/as[1-6]*
: Temporary storage files

SEE ALSO

adb(1), ld(1), m4(1), nm(1), strip(1)

a.out(4) in *A/UX Programmer's Reference*

"as Reference" in *A/UX Programming Languages and Tools, Volume 1*

NAME

asa — interprets ASA carriage control characters

SYNOPSIS

asa [*file*]...

ARGUMENTS

file Specifies the output of a Fortran program that uses ASA carriage control characters. If you do not specify *file*, asa reads the standard input.

DESCRIPTION

asa interprets the output of Fortran programs that use American National Standards Institute (ANSI) carriage control characters to print on a line printer.

The asa command assumes that the first character of each line in *file* is a control character and transforms the control character into a printer control character. Here are the control characters and the actions that asa takes:

space Inserts a single newline character before the line, where *space* is the ASCII character 020.

0 Inserts two newline characters before the line.

1 Inserts a character that causes an advance to a new page before the line.

\+ Inserts a character that causes the previous line to be overprinted.

The asa command causes the first line of each input file to be printed on a new page. If any line does not begin with a control character, asa passes on the second and subsequent characters of the line and writes asa: 1 invalid input lines in *file* on the standard error.

EXAMPLES

This command line uses asa as a filter on the output of a Fortran program called a.out and prints the output of asa by using the lp command, whose default printer is a line printer:

```
a.out | asa | lp
```

This command lets you see the output of a Fortran program that has been sent to a file:

```
asa file
```

FILES

/bin/asa
Executable file

SEE ALSO

efl(1), f77(1), fpr(1), fsplit(1)

NAME

at, batch — run commands at a later time

SYNOPSIS

at *time* [*day*] [+ *increment*]

at -l [*job-number*]...

at -r *job-number*...

batch

ARGUMENTS

+ *increment*
Specifies an optional *increment* that further specifies the time at which to run your job. The increment is a number suffixed by one of the following times: minutes, hours, days, weeks, months, or years. An example of an increment is + 1 month as in at 0815pm + 1 month. The at command also accepts the singular form of each increment.

day Specifies an optional day on which the command is to be run. The day can be a month name followed by a day number (for example Jan 29); a month name followed by a day number, a comma, and a year number (for example, Jan 29, 1991); or a day of the week, spelled out or abbreviated to three characters (for example, Tuesday or Tue). If you specify a month that is less than the current month and you do not specify a year, at assumes the next year. Two special days, today and tomorrow, are recognized. If you do not specify a date, at uses today if the specified hour is greater than the current hour or tomorrow if it is less.

-l [*job-number*]...
Reports by job number the at and batch jobs that you currently have scheduled.

-r *job-number*
Removes the specified *job-number*, which was previously scheduled by means of at or batch. You can remove only your own jobs unless you are logged in as root.

time
Specifies the time at which to run your job. You must specify a *time* argument, which can be one, two, or four digits. The at command interprets a time of one or two digits as specifying an hour of the day and interprets a time of four digits as specifying the hour and the minute. You can also specify the time as two numbers separated by a colon, meaning *hour*:*minute*. You can append am or pm to *time*. Otherwise, at uses a 24-hour clock. You can also append zulu to

indicate Greenwich mean time. The at command also recognizes a special time of noon, midnight, now, or next.

DESCRIPTION

at and batch read commands from the standard input and place them on a queue to be run at a later time by the atrun command, which uses the Bourne shell (/bin/sh) to run the job. The at command allows you to specify when the commands are to be run, while jobs queued with batch are run when system load level permits. When the commands are run, atrun mails to you the standard output and standard error output of the job unless you redirected the output elsewhere.

You can use at and batch if your name appears in the file /usr/lib/cron/at.allow. If that file does not exist, at checks the file /usr/lib/cron/at.deny to determine if you should be denied access. If neither file exists, only a user who is logged in as root can submit a job. These files consist of one login ID per line.

The at and batch commands read the standard input to get the command to run. To terminate the entry of commands, press CONTROL-D. To indicate success, at and batch write the job number and scheduled time on standard error.

When the job is run, it inherits the shell environment variables, current directory, file-creation mask, and file-size limit that were in effect when you entered the at or batch command. The job does not inherit your open file descriptors, traps, or priority.

If the system is not running at the scheduled time and is started later, atrun does not run the job.

EXAMPLES

Here are some valid at commands:

```
at 0815pm Jan 24
at 8:15pm Jan 24
at now + 1 day
at 5 pm Friday
```

Use this sequence at a terminal to redirect standard output:

```
batch
nroff filename > outfile
```

The following sequence, which demonstrates how to redirect standard error to a pipe, is useful in a shell script. The order of output redirection specifications is significant.

```
batch <<!
nroff filename 2>&1 > outfile | mail login-id
```

!

To have a job reschedule itself, invoke `at` from within a shell script by including code similar to the following line within the script:

```
echo "sh script" | at 1900 thursday next week
```

STATUS MESSAGES AND VALUES

`at: you are not authorized to use at. Sorry.`
Indicates that your name is not in `/usr/lib/cron/at.allow` or that your name is in `/usr/lib/cron/at.deny`.

`at: bad date specification`
Indicates that the *day* or *time* argument is incorrect.

`warning: commands will be executed using /bin/sh`
Indicates that the command was successful.

FILES

`/usr/bin/at`
Executable file

`/usr/bin/batch`
Executable file

`/usr/lib/atrun`
Executable file, invoked by `cron`, that runs the jobs in the `at` and `batch` queues

`/usr/lib/cron/at.allow`
File containing a list of allowed users

`/usr/lib/cron/at.deny`
File containing a list of denied users

`/usr/lib/cron/queuedefs`
File containing scheduling information

`/usr/spool/cron/atjobs`
Directory containing the job queue

SEE ALSO

`crontab`(1), `kill`(1), `mail`(1), `nice`(1), `ps`(1), `sh`(1)

`cron`(1M) in *A/UX System Administrator's Reference*

NAME

atlookup — looks up network-visible entities (NVEs) registered on the AppleTalk network system

SYNOPSIS

atlookup [-d] [-r *nn*] [-s *ss*] [-x] [*object*[: *type*[@*zone*]]]

atlookup -z [-C]

ARGUMENTS

-C Prints zones in multiple columns.

-d Prints the network address in decimal numbers.

object
Specifies the name of the object to be looked up.

-r *nn*
If the lookup is unsuccessful, the system tries again the number of times specified by *nn*. The default is to try the lookup eight times.

-s *nn*
Instructs atlookup to wait a certain number (*ss*) of seconds between consecutive attempts to complete a lookup successfully. The default is to space retries one second apart.

type Specifies the type of the object to be looked up.

-x Prints the 8-bit ASCII characters on output as hexadecimal numbers of the form (where *X* is a hexadecimal digit). This option is useful when you are using a terminal other than the A/UX system console.

-z Lists all zones in the network.

zone
Specifies the zone in which the lookup is to be performed. You can use an asterisk instead of a zone name to indicate the current zone name. If you don't specify a zone name, the current zone is the default.

The *object* and *type* arguments can contain wildcard characters. The equal sign (=) indicates a wildcard lookup. For wildcard lookups to work correctly with all nodes, the only character specified in the string must be the wildcard character. However, AppleTalk Phase 2 nodes also honor a single embedded wildcard character, '='. Under this scheme, one wildcard character can appear anywhere in the string and can match zero or more characters. Note, however, that although an embedded '=' is acceptable in *object* and *type* arguments of atlookup, only the nodes implementing AppleTalk Phase 2 protocols respond to such a query. For this reason, the resulting list of NVEs may be incomplete.

DESCRIPTION

`atlookup` uses the Name Binding Protocol (NBP) to look up names and addresses of the specified NVEs.

The default is to look up all the entities (of all types) in the current zone. Specifying the object, type, or zone on the command line changes the scope of lookup.

Information about the NVEs is displayed in a table format, one line per NVE. Each line gives the names of the object, type, and zone and the numbers of the network, node, and socket.

EXAMPLES

This command looks up all NVEs registered in the local AppleTalk zone:

```
atlookup
```

In response, the system displays output similar to this:

```
Found 5 entries in zone My-Zone
6b5b.c3.ea 3-Eyed Monster:LaserWriter
6b5b.80.fd 3-Eyed Monster Spooler:LaserWriter
6b14.84.ea Incognito :LaserWriter
6b19.a3.fd Light of Day:AFPServer
6b51.27.fd Nets-R-Us Spooler:LaserWriter
```

In an extended AppleTalk network, this command displays all NVEs (of any type) in the current zone whose names start with *L* and end in *y*:

```
atlookup L=y:=
```

The output might be similar to this:

```
Found 1 entries in zone My-Zone
6b19.a3.fd Light of Day:AFPServer
```

FILES

`/usr/bin/atlookup`
Executable file

SEE ALSO

`at_cho_prn`(1), `atprint`(1), `atstatus`(1)

Inside AppleTalk

NAME

atprint — transfers data to a printer by using AppleTalk protocols

SYNOPSIS

atprint [*printer-name*[: *printer-type*[@*zone*]]]

ARGUMENTS

printer-name

Specifies the name of the printer you want to use.

printer-type

Specifies the type of printer, such as LaserWriter or ImageWriter. Use this option when you want to allow the network to select the printer, but only a printer of a given type. If you omit this option, LaserWriter is the printer type used by default.

For example, when the printer name is specified with wildcards. (See atlookup(1) for an explanation of wildcards.) The print device used is the one chosen by the network. By supplying LaserWriter as the printer type in a case such as this, you can restrict the network to choosing a printer that can handle PostScript instructions.

The full range of possible replacement values for *printer-type* depends on the configuration of your network. Each different type of print device broadcasts its printer-type and printer-name identification when it registers itself with the network. You can use atlookup to obtain a report showing this information for all the AppleTalk devices on your network (see atlookup(1)).

zone

Specifies the AppleTalk zone in which the printer resides. If you omit this argument or specify it as *, the local zone is used.

DESCRIPTION

atprint uses a printing protocol to establish a connection to an AppleTalk device, where it sends data received on its standard input until it reaches an *end-of-file* character. When it detects an end-of-file character, atprint closes the AppleTalk session with the device, enabling other users to gain access to the printer.

You can select the destination AppleTalk device through the command-line arguments as described in the ''Arguments'' section earlier in this manual page. If you do not specify any of these arguments, atprint uses the printer that was last selected either with the Chooser or with at_cho_prn (see at_cho_prn(1)).

Often the printer you access by way of an AppleTalk connection is a LaserWriter. Many LaserWriter models are PostScript printers. If you are using such a LaserWriter, the data that you send it must already be

translated into the PostScript page-description language. For example, the `psdit` command translates the output from `troff` (invoked with the `-Tpsc` option) into PostScript:

```
troff -Tpsc -mm file | psdit |atprint
```

The `atprint` command displays one or more messages indicating the AppleTalk device with which it is communicating and possibly many device status messages (such as when another print job is occupying the printer for a period of time). In the preceding example, the default printer is used. (See the ''Arguments'' section earlier in this manual page.)

(Note that the `atprint` command does not honor requests from a LaserWriter regarding the downloading of fonts. Likewise, it does not prepend a PostScript header to the data stream in the same manner as the printer drivers in the Macintosh Operating System. In the preceding example, a PostScript header is still provided because `psdit` prepends its own header as part of the PostScript conversion process.)

In AppleTalk programming terms, the arguments make up a network-visible entity (NVE), where

printer-name[:*printer-type*[@*zone*]]

corresponds to the AppleTalk object, type, and zone:

object:*type*@*zone*

EXAMPLES

This command line maps a plain text file into PostScript and then submits it to `joe's` printer:

```
enscript -p- file | atprint "joe's printer"
```

WARNINGS

The `atprint` command does not process the input files as does `lpr`. To print ASCII files properly on a PostScript printer with `atprint`, you must preprocess the files with `pstext` or `enscript`. Likewise, you must preprocess files produced by `troff` with `psdit`(1).

FILES

`/usr/bin/atprint`
Executable file

SEE ALSO

`at_cho_prn`(1), `atlookup`(1), `atstatus`(1), `enscript`(1), `lpr`(1), `psdit`(1), `pstext`(1)

''AppleTalk Programming Guide'' in *A/UX Network Applications Programming*

''Administering AppleTalk'' in *A/UX Network System Administration*
Inside AppleTalk

NAME

atstatus — displays status information from an AppleTalk device

SYNOPSIS

atstatus [*object* [:*type* [@*zone*]]]

ARGUMENTS

object

Specifies the name of the AppleTalk device. Wildcard characters are not permitted. If you don't specify the AppleTalk device, atstatus uses the system default. If the name contains spaces, put quotation marks around the name. Here is an example:

```
atstatus "Sharon's Print Shop"
```

type Specifies the type of server. If you don't specify the *type* argument, the default is LaserWriter. If you supply a *zone* argument, you must also supply a *type* argument.

zone

Specifies the zone in which the AppleTalk device resides. If you don't specify the *zone,* the system defaults to *, your local zone.

DESCRIPTION

atstatus gets the status string from an AppleTalk device, such as a LaserWriter.

FILES

/usr/bin/atstatus

Executable file

SEE ALSO

at_cho_prn(1), atlookup(1), atprint(1)

Inside AppleTalk

NAME

at_cho_prn — allows you to choose a default printer on the AppleTalk internet

SYNOPSIS

at_cho_prn [*type*[@*zone*]]

ARGUMENTS

type[@*zone*]

Specifies the type of printer to be used, and the area (*zone*) in which it resides. If you don't use the *type* argument on the command line, at_cho_prn displays all entities of the types LaserWriter and ImageWriter. The system prompts you to select a printer by entering the appropriate number from the printer list display. If you don't enter the *zone* part of the argument on the command line, at_cho_prn lists all the zones in the internet and prompts you to choose the zone in which you'd like to select your default printer.

DESCRIPTION

The at_cho_prn command displays a list of printer selections and saves the name of the printer that you select. The at_cho_prn command checks the network to determine which printers are registered on that network.

After you specify the zone, at_cho_prn lists the printers (of type *type*) available in that zone.

EXAMPLES

The command

```
at_cho_prn 'LaserWriter@*'
```

produces output similar to this:

```
ITEM  NET-ADDR     OBJECT : TYPE
  1: 56bf.af.fc AnnLW:LaserWriter
  2: 56bf.ac.cc TimLW:LaserWriter

ITEM number (0 to make no selection)?
```

where NET-ADDR is the AppleTalk internet address (printed in hexadecimal) of the printer's listener socket, and OBJECT:TYPE is the name of the registered printer and its type.

FILES

/usr/bin/at_cho_prn

Executable file

SEE ALSO

`atlookup`(1), `atprint`(1), `atstatus`(1)

Inside AppleTalk

"Administering AppleTalk," in *A/UX Network System Administration*

"AppleTalk Programming Guide," in *A/UX Network Applications Programming*

NAME

awk — scans a file for lines that match a specific pattern

SYNOPSIS

awk [-F*field-separator*] *'pattern-action... '* [[-v] *variable=value*]... [*file*]...

awk [-f *awk-source-file*] [-F*field-separator*] [[-v] *variable=value*]... [*file*]...

ARGUMENTS

-f *awk-source-file*
: Specifies the file containing the instruction that awk should interpret.

-F*field-separator*
: Specifies the character to be treated as the field separator when awk parses a record into fields.

file
: Specifies the file or files containing text data to be processed by awk.

pattern-action
: Specifies an awk instruction, which is provided in the form of a pattern followed by an action enclosed in braces:

pattern {*action*}

[-v] variable=value
: Specifies the value of an awk variable that is established for use in the main sections of an awk program, which consists of any number of *pattern-action* arguments. If the -v option is present, the variable is also available in the BEGIN (initialization) section of an awk program.

DESCRIPTION

awk effectively handles most programs containing text-parsing, report generation, and record validation tasks. These programs typically contain a brief list of instructions that specify text-scanning and text-manipulation functions.

The standard operation of awk is to scan each input file once, looking for matches between each input record and any of a set of patterns that you supply. These pattern instructions are accompanied by action instructions. Sometimes the action instructions merely establish settings that affect text processing that is undertaken by awk as part of its standard operation, such as the parsing of records into fields.

So that text patterns can be sought in specific positions in an input record, awk splits the input record into fields at every occurrence of a *field-separator* character. After an input record is split into fields, each field is assigned to a field variable, such as $1, $2, $3, and so forth. These

variables can be used to reference input fields either in the pattern or the action portion of a *pattern-action* argument.

You can obtain a measure of control over the field-parsing function by specifying your own field separator for parsing purposes. The default field separator is white space (tabs or spaces). You can change this separator by making a different assignment to the variable `FS`, or through the command line by specifying a *field-separator* character along with the `-F` option. To ensure that your own field separator takes effect before any input records are parsed into fields, use the `-F` construct or place the assignment in an action associated with the `BEGIN` pattern. (See the example at the end of ''Patterns,'' later in the ''Description'' section.)

(A regular expression can also be assigned to the `FS` variable, in which case the field delimiter can be any one of the possible values that match the regular expression.)

Although it looks like a field reference, `$0` refers to the entire input record, with field delimiters unstripped.

For the purposes of documenting syntax, a pattern and its associated actions are considered one *pattern-action*. As shown in the first syntax description in the ''Synopsis'' section, *pattern-action* arguments can be supplied directly on the command line. Alternately, you can specify the `-f` option so that *pattern-action* arguments can be placed inside of an `awk` program file, as shown in the second syntax description (see SYNOPSIS). In the latter case, replace *awk-source-file* with the name of the program file with the `awk` instructions you want to use.

Any time an input record contains a substring that is sought as specified by *pattern*, `awk` performs the associated action. The text of an input record that is matched by a pattern can be accessed easily through references to the variables `$0`, `$1`, `$2`, and so forth.

Input records can be acted upon immediately or handled less directly. An example of an immediate action is the printing of the contents of a matching input record as soon as it is encountered. An example of a less immediate action is storing a record in a variable when it is first encountered, then printing it later if later conditions warrant it, such as when the contents of subsequent records invalidate it and an error message is desired.

A stored value persists until it is changed by another portion of the same *pattern-action* or by an entirely different *pattern-action*. Such assignments permit actions to be gated not only by the text of the input record being scanned but also through the stored text drawn from previous input records.

Command-Line Options

Either *pattern-action* arguments are specified inside the awk command lines as shown in the first syntax description line, or they are supplied in a file through specification of file arguments along with the -f option, as shown in the second syntax description line. When *pattern-action* arguments all appear in the command line, they should be formed into one string enclosed in single quotation marks ('). The quotation marks protect them from being interpreted by the shell. Refer to the awk chapter in *A/UX Programming Languages and Tools, Volume 2* for more information about shell and awk cooperation.

The level of escapement afforded by the single quotation marks causes any references to shell variables to remain unsubstituted by the shell. To enable their substitution requires the use of awk variables that assign values inside the command line.

Variables that are initialized on the command line provide a means of passing parameter values between the shell and awk. The most common use for passed parameters is to access the values of positional variables available from within shell scripts ($1, $2, and so forth). The format of these assignments is similar to that of variable assignments, except that an unescaped space cannot be used on either side of the equal sign, as follows:

```
awk -f awkfile datafile variable1=x variable2=$1
```

If the parameter assignment is preceded by a -v option, the value so assigned is made available even in the BEGIN (initialization) section of the awk program. Otherwise, the value is not assigned to the variable until after the BEGIN section has been evaluated.

Like input files, the passed parameters are also evaluated in the order in which they appear: Passed parameters that are specified after an input file will not be available while the system is processing that input file. Passed parameters that are specified before any number of input files will be available when processing those input files.

If no input file is specified, the standard input is read until exhausted. When several input files are specified, they are read in the order in which they are specified. If the shorthand notation for standard input (-) is specified as one of several *file* arguments, the standard input is also read in the order in which it is specified.

Patterns

The pattern portion of a *pattern-action* argument often involves the scanning of text for occurrences of a particular text pattern. These patterns are specified through a pattern-seeking template, better known as a regular expression. For a more detailed explanation of regular expressions, refer to ed(1).

Regular expressions must be surrounded by slashes. The format for a regular expression is

/character-col1... character-colN/

where *character-col1* through *character-colN* represent the first through last characters to seek before a substring is considered ''matched.''

Besides supplying a normal character to replace *character-col1* and other character positions, you can use a special or wildcard character, such as the period, which matches any character at that position. An asterisk matches any number of any characters from that position onward. Other special characters are the caret (^) and dollar sign ($), which ''match'' the beginning of a line and the end of a line, respectively. The only sensible place to insert the caret is at the beginning of *pattern*. Likewise, the only sensible place to insert the dollar sign is at the very end of *pattern*.

Besides supplying a single character to replace *character-col1* and other character positions, you can supply a character range or a character list enclosed in brackets. Thus,

```
/^[A-Z][aeiou]/
```

evaluates as true for all input records that start with an uppercase character followed by a vowel.

The pattern portion of the *pattern-action* argument can be any expression, including ones that do not involve pattern-seeking. For example,

```
$1 > 0  { print }
```

is a valid *pattern-action* argument that prints all input records with a first field that is greater than 0.

Pattern expressions often test for the presence of certain text patterns, either within the entire input record or within one or more fields in an input record. Field-scoped searches require one of the ''pattern-seeking'' operators and a regular expression, as follows:

```
$0 ~ /Employee/   { action... }
$3 ~ /Employee/   { action... }
```

If you search the entire input record for matching strings, you do not have to supply the `$0 ~` portion of the line, since this portion will be assumed when a regular expression is supplied by itself as the pattern. This convention makes the following patterns equivalent:

```
$0 ~ /Employee/
/Employee/
```

To seek a contiguous set of input records starting from a record that matches *pattern1* and ending with a record that matches *pattern2*, specify two regular expressions separated by a comma, as follows:

/pattern1/ , / pattern2/ { action... }

The action is performed for all input records between an occurrence of the first pattern and the next occurrence of the second pattern.

The special patterns `BEGIN` and `END` can be used to establish actions to be taken before the first input record is read and after the input stream is exhausted. For example, a tab can be made the field separator (exclusively) with

```
BEGIN { FS = "\t" }
```

Actions

A *pattern-action* argument has the form

pattern { *action* }

A missing {*action*} argument triggers the printing of matching input records; a missing *pattern* argument causes the associated action to be performed for every input record (as if every input record matched the missing pattern). An *action* argument is a sequence of statements. A statement can be one of the following code fragments:

```
if ( conditional ) statement [ else statement ]
while ( conditional ) statement
for ( expression ; conditional ; expression ) statement
break
continue
{ [ statement ]... }
variable = expression
next
exit
```

Statements are terminated by semicolons, newline characters, or right braces. Expressions take on string or numeric values as appropriate and are built with the operators `+`, `-`, `*`, `/`, `%`, and ''concatenation'' (indicated by a blank). The C operators `++`, `--`, `+=`, `-=`, `*=`, `/=`, and `%=` are also available in expressions. Variables can be scalars, array elements (denoted *x*`[`*i*`]`), or fields. Variables are initialized to the null string. Array subscripts can be any string, including strings generated automatically when numeric expressions are used as subscripts. String constants are enclosed in double quotation marks (").

The next and exit functions affect control flow. Use exit to terminate processing without any further actions. Use next to terminate any remaining actions that would have been gated for the current input record, skipping to the beginning of the current *awk-source-file* so that processing can continue with the next input record.

Output Functions

The output functions include the following statements:

print [*expression*] [[,] *expression*]...
printf (*format-string, expr* [, *expr*]...)

Both of these statements can print to files as well as the standard output, as described by the more general syntax

print-command [>*file*]

Use the print statement to print the results of *expression* arguments followed by the output record separator character given by the variable ORS. If print is specified without any accompanying arguments, the entire input record is printed. If several expressions are supplied, separated by commas, the result of each expression is printed, separated by the output field separator given by the variable OFS. See ''Built-in Variables'' later in the ''Description'' section for more built-in variables.

Use the printf statement to format and print the result of *expr* arguments in accordance with *format-string* (see printf(3S)). Another way to place data on the awk output stream is to use the system function

system(*expression*)

In this case, *expression* must compute to a valid shell command so that it can be executed outside the context of awk. Any output resulting from the execution of the command is inserted into the output of awk. This function returns the exit status for the command so that you can test for successful execution by testing for a 0 exit value. (This is the case for most, but not all, commands.)

Input Functions

Besides being supplied as command-line arguments, multiple input files are supported through the getline function. This record-reading function can be one of the actions associated with a BEGIN or an END pattern, as well as any other patterns. A typical use is to associate this action with the BEGIN pattern to initialize the contents of an array from static data stored in an external file. Since the return value is 1 as long as the input file is not exhausted, you can use the following code fragment to establish the file *table*:

```
BEGIN {
```

```
    while ( getline array[count] <"table" > 0 )
    { count = count + 1 }
}
.
.
.
```

This command can be specified in any of four different forms:

```
getline
getline variable
getline <file
getline variable <file
```

The first form reads the next input record. Unlike the `next` statement, with this form control remains at the place where `getline` occurs within the current *pattern-action* argument and proceeds to any *pattern-action* arguments that follow, until the *end-of-file* character is reached.

The second form behaves in the same way except that certain variables (`$0`, `$1`, and so forth) are not reset and the content of the input record is assigned to *variable* unstripped of field separators.

The third and fourth forms are the same as the first and second forms except that the input record is read from *file*. If *file* is an explicit reference to a file, enclose it in quotation marks to make it a string constant. (Otherwise it is likely to be interpreted as a variable that is dynamically initialized to an empty string.) To switch between many different input files, use the `close(`*file*`)` function before opening any new files for reading.

Other String Functions

Here are the built-in functions for strings:

`index(`*string1*`,` *string2*`)`
: Returns the index at which *string2* first occurs inside *string1* or 0 if there is no match.

`length(`*string*`)`
: Returns the length of its argument taken as a string, or of the whole input record if no argument is supplied.

`match(`*string*`,` *pattern*`)`
: Returns the index at which the regular expression *pattern* first occurs inside *string* while setting the variables `RSTART` and `RLENGTH`. Returns 0 if there is no match.

`split(`*string*`,` *array*`,` *separator*`)`
: Splits *string* into fields that are assigned to elements in *array* with subscripts `1`, `2`, and so on. A new field is created at each occurrence

of *separator* within *string*. It returns the number of fields that were parsed.

`substr(`*string*`,` *position*`,` *length*`)`
Returns the *length*-character substring of *string* that begins at position *position*.

`sprintf(`*format-string*`,` *expr*`[,` *expr*`]...)`
Formats expressions in accordance with *format-string* (described in `printf`(3S)), returning the resulting string.

`sub(`*pattern*`,` *replacement*`[,` *variable*`])`
`gsub(`*pattern*`,` *replacement*`[,` *variable*`])`
Performs text substitution (search-and-replace) functions either for the first matched substring (`sub`) or globally for every matched substring (`gsub`).

Number Functions

Here are the built-in functions for numbers:

`atan2(`*y*`,`*x*`)`
Returns the arctangent of *y*/*x* in radians in the range $-\pi$ to π.

`cos(`*radians*`)`
Returns the cosine of the angle measure.

`exp(`*power*`)`
Returns *e* raised to the *power* power.

`int(`*real*`)`
Truncates *real*, returning an integer.

`log(`*x*`)`
Returns the natural logarithm of *x*.

`rand()`
Returns a pseudo-random number between 0 and 1.

`srand([`*seed*`])`
Sets the seed for the random number generator to *seed* or to the time of day if *seed* is missing.

`sin(`*radians*`)`
Returns the cosine of the angle measure.

`sqrt(`*x*`)`
Returns the square root of *x*.

User-Defined Functions

User functions can be called just as built-in functions are, once they are declared with

`function` *name*`(`*arg*`...) { `*body*` }`

Within *body*, the function `return(`*expression*`)` can be used to cause the user function to return the value of the supplied expression.

Expressions

This discussion of expressions applies within action statements and within patterns. Only certain action statements can include expressions; refer to ''Actions,'' earlier in the ''Description'' section for more information. Parentheses can be used to establish operation precedence for expressions containing several operators.

Expressions can be string or number constants, variables, or field references as well as combinations of these joined by equal (==), not equal (!=), greater-than (>), less-than (<), greater-than-equal (>=), and less-than-equal (<=). Because they produce Boolean results (true or false), two or more of the preceding comparison operations can be related by means of Boolean operators: logical AND (&&), logical OR (||), and NOT (!).

To test for the existence of various substrings in a string, specify the string followed by one of the pattern-seeking operators (~ and !~) followed by a regular expression. Use ~ to test whether the string contains a substring that is sought by the regular expression supplied. Use !~ to test whether the string does not contain a substring that is sought by the regular expression supplied.

The following example uses all of these types of operations:

```
{ if ( NR > 1  && $0 ~ /+/ ) print }
```

In the next line of code, which is equivalent to the one just given, the operations have been moved into the pattern area:

```
$0 ~ /+/  &&  NR > 1  { print }
```

No operation exists specifically to request conversions between numbers and strings, or between strings and numbers. To force an expression to be treated as a number, add 0 to it; to force it to be treated as a string, concatenate the null string ("") to it.

Built-in variables

Other variable names with special meanings include

`NF` the number of fields in the current record

`NR` the ordinal number of the current record

`FNR`
: the ordinal number of the current record relative to the beginning of the current input file

`FILENAME`
: the name of the current input file

OFS
: the output field separator (blank by default)

ORS
: the output record separator (the newline character by default)

OFMT
: the output format for numbers (`%.6g` by default)

ARGC
: a variable that is set to the total number of command-line arguments that were offered on the `awk` command line

ARGV[]
: a built-in array that is set to the command name (`awk`) at index `0`, the first command-line argument at index `1`, and so on up to the last command-line argument at index *n*

Overview of `awk` Processing and Preprocessing

For each input record, `awk` performs the ''matched'' *pattern-action* operations. Thus, the actions that `awk` performs usually vary with each input record. The effect is similar to that of creating a number of different programs, where each one is a particular accumulation of lines from a master collection. Each of the accumulated subprograms is run whenever its triggering records show up in the input stream, possibly many times over. Through careful selection of patterns, these subprograms can be closely tailored to the kind of data that is present in the input record.

When the input data is not already partitioned nicely into fields and records, the use of preprocessing can be useful to transform the data into more regular units from which meaning is more easily extracted. For text data that already contains field separators, the field values that indicate variant records are easily detected when they can be expected at a fixed field location references within patterns. (See ''Patterns,'' earlier in the ''Description'' section.) For data that is not already subdivided or regularized, preprocessing with `sed` or `awk` is often desirable so that units of data that affect the meaning of other units of data can be incorporated into the same record, or so that independently meaningful units of data are separated into new records.

When you are combining spans of data into the same record, it is often desirable to place context-establishing data at the beginning so that certain patterns can be sought in certain positions by using the corresponding features of regular expressions, such as the caret (^).

In cases involving irregular data, the preprocessing concern of greatest import is the generation of appropriate record and field boundaries within the data. For instance, each pass of preprocessing can be designed so that a particular output field (or a particular record within a set of records) will be

set to an appropriate value for identifying the context of a certain amount of data. For example, the nesting of procedures inside braces is more easily unraveled if the beginning and ending braces always occupy the first field of an input record, or a dedicated input line.

EXAMPLES

The following command prints lines from the file `data` that are longer 72 characters:

```
awk "length > 72" data
```

The following command prints the first two fields of each line in reverse order:

```
awk '{ print $2, $1 }' filea
```

prints the first two fields of each line in reverse order.

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

adds up the first column and prints the sum and average.

```
awk '{ for (i = NF; i > 0; --i)
           print $i }' filea
```

prints all the fields of each line in reverse order. The fields are printed one per line in this example.

```
awk "/start/, /stop/" filea
```

prints all records between start/stop-pattern pairs for every such pair in the file.

```
awk '
$1 > max  { max = $1 }
END       { print "Max field 1 value=" max }'
```

prints the maximum value that appears in field 1 of each input record.

FILES

`/bin/awk`
Executable file

SEE ALSO

`grep`(1), `lex`(1), `sed`(1)

"`awk` Reference," in *A/UX Programming Languages and Tools, Volume 2*

The awk Programming Language by A.V. Aho, B.W. Kernighan, and P.J. Weinberger (Reading, MA: Addison-Wesley, 1988)

NAME

banner — generates a poster

SYNOPSIS

banner *string...*

ARGUMENTS

string

Specifies a string of no more than ten characters. The banner command truncates any additional characters. If you use quotation marks to enclose words separated by a space, banner generates its output on one line. Otherwise, banner interprets multiple words as separate arguments and puts each word on its own line.

DESCRIPTION

banner reproduces its arguments in large letters, using the number sign (#), on the standard output.

EXAMPLES

This command causes the characters H, A, P, P, and Y to be printed in large letters:

```
banner HAPPY | lp
```

FILES

/usr/bin/banner

Executable file

SEE ALSO

banner7(1), echo(1)

NAME

banner7 — generates a large banner

SYNOPSIS

banner7 [-w [*width*]] [*text*]

ARGUMENTS

text Specifies the text of the banner. If you omit *text*, banner7 displays a Message: prompt and reads its standard input until you press RETURN. If you type more than 255 characters, banner7 truncates the 256th and remaining characters.

-w[*width*]
Specifies the width of the output. If you omit *width*, banner7 uses 80 columns. If you do not specify this option, banner7 uses a default of 132 columns.

DESCRIPTION

banner7 generates a large banner, using the number sign (#) to form the characters, on the standard output. Typically, you pipe the output of banner7 to the lp command for printing. The output of banner7 is designed to be printed on a line printer.

LIMITATIONS

The banner7 command cannot handle these ASCII characters:

```
< > [ ] \ ^ _ { } | ~
```

Also, banner7 substitutes an alternative representation for a quotation mark ("), an apostrophe ('), and an ampersand (&) that is funny looking but still useful.

The -w option is implemented by skipping some rows and columns. As you narrow the column width, the output becomes grainy and letters occasionally run together.

STATUS MESSAGES AND VALUES

The message:

```
The character 'c' is not in my character set
```

is produced to indicate a character that banner7 cannot handle.

FILES

/usr/bin/banner7
Executable file

SEE ALSO

banner(1), echo(1)

NAME

basename, dirname — get part of a pathname

SYNOPSIS

basename *string* [*suffix*]

dirname *string*

ARGUMENTS

string

Specifies an absolute or a relative pathname.

suffix

Specifies an optional suffix that, if present in *string*, basename is to remove.

DESCRIPTION

basename examines *string* for the last slash (/) and returns the characters that follow the slash. For example, basename /usr/bin/vi yields vi.

dirname examines *string* for the last slash and returns the characters that precede the slash. For example, dirname /usr/bin/vi yields /usr/bin.

Both basename and dirname write on the standard output. Neither command verifies that *string* is a valid pathname on the current system.

EXAMPLES

The basename and dirname commands are most commonly used in shell scripts where the commands are enclosed within back quotes (‘). Enclosure within backquotes causes the command to be executed and the result substituted as an argument to another command. For example, if $1 is /users/tom/prog/mine.c, this sequence compiles /users/tom/prog/mine.c and moves the output to a file named mine in the current directory:

```
cc $1
mv a.out 'basename $1 '.c''
```

This sequence sets the Bourne shell variable NAME to /users/tom/prog:

```
NAME='dirname /users/tom/prog/mine.c'
```

STATUS MESSAGES AND VALUES

If *string* is only a slash, an error results and basename displays this message:

```
expr: syntax error
0
```

FILES

`/bin/basename`
Executable file

`/bin/dirname`
Executable file

SEE ALSO

sh(1)

See at(1)

NAME

bc — processes an arbitrary-precision arithmetic language

SYNOPSIS

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

ARGUMENTS

-c Causes bc not to invoke dc. If you specify this option, bc output that is normally sent as input to dc is sent to the standard output instead.

file Specifies a file that contains statements that bc can interpret. You can use a file argument to set built-in names, such as scale.

-l Causes bc to use an arbitrary-precision math library.

DESCRIPTION

bc is an interactive processor for a language that resembles C but provides unlimited-precision arithmetic. Actually, bc is a preprocessor for the dc command, which it invokes automatically.

The bc command reads the standard input, but you can put bc commands in a file that bc reads when it starts up.

The section that follow describe the syntax of the bc language, where *name* is a variable or function name, *expression* is an expression, and *statement* is a statement.

Comments

Comments begin with a slash and an asterisk (/*) and end with an asterisk and a slash (*/).

Names

You can construct variable names with the letters a through z. Uppercase letters are not allowed. You can reference array elements by using square brackets, as in *name*[*expression*]. You can also use these built-in names:

ibase
: Sets the input number radix.

obase
: Sets the output number radix.

scale
: Sets the number of digits that are retained to the right of the decimal point after an arithmetic operation.

auto
: Defines variables that are pushed down during function calls.

You can use the same name for an array, a function, and a simple variable simultaneously.

All variables are global to the program. When using arrays as function arguments or defining them as automatic variables, you must put empty square brackets after the array name.

Other Operands

Other operands are constructed from arbitrarily long numbers with an optional sign or an optional decimal point. Here are some examples:

`(`*expression*`)`
Evaluates the value of *expression*.

`sqrt(`*expression*`)`
Returns the square root of *expression*.

`length(`*expression*`)`
Returns the number of significant decimal digits in *expression*.

`scale(`*expression*`)`
Returns the current value of `scale`.

`abc (`*expression* `, ... ,` *expression*`)`
Calls the function `abc` with the specified arguments.

Operators

Here are the available operators:

```
+   -   *   /   % ^
++   --
==   <=   >=   !=   <   >
=   =+   =-   =*   =/   =%   =^
```

The percent sign (`%`) is the modulo operator, and the caret (`^`) is the exponentiation operator. You can use a double plus sign (`++`) and a double minus sign (`--`) as prefix and postfix operators on names.

Statements

Here are the forms that a statement can take:

expression

`{`*statement* `;` `...` `;` *statement*`}`

`if (`*expression*`)` *statement*

`while (`*expression*`)` *statement*

`for (`*expression*`;` *expression*`;` *expression*`)` *statement*

`break`

`quit`

The value of a statement that is an expression is printed unless the main operator is an assignment.

You can separate statements by using a semicolon or a newline character. Two newline characters cannot follow a left brace ({).

Function Definitions

Here is the form of a function definition:

```
define name (name, ..., name) {
        auto name, ... , name
        statement; ... statement
        return (expression)
}
```

The bc command passes all function arguments by value.

Math Library

These functions make up the math library that becomes available when you use the -l option:

s(*x*)	sine
c(*x*)	cosine
e(*x*)	exponential
l(*x*)	log
a(*x*)	arctangent
j(*n*,*x*)	Bessel function

EXAMPLES

This sequence defines a function that computes an approximate value for the exponential function:

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

This sequence calculates approximate values of the exponential function of the integers 1-10:

```
for(i=1; i<=10; i++) e(i)
```

LIMITATIONS

The `bc` command supports neither the logical AND operator (`&&`) nor the logical OR operator (`||`).

The `for` statement must have all three expressions.

The `quit` statement is interpreted when it is read rather than when it is executed.

FILES

`/usr/lib/bc`
: Executable file

`/usr/bin/dc`
: Executable file that `bc` calls

`/usr/lib/lib.b`
: Mathematical library file invoked by the `-l` option

SEE ALSO

`dc`(1)

"`bc` Reference" in *A/UX Programming Languages and Tools, Volume 2*

NAME

bdiff — compares the difference between two large files that are too big for diff to handle

SYNOPSIS

bdiff *file1 file2* [*lines-per-segment*] [-s]

ARGUMENTS

file1

Specifies the file that will be compared with *file2*. If this file is a hyphen (-), bdiff reads the standard input. In this case, *file2* cannot be a hyphen.

file2

Specifies the file that will be compared with *file1*. If this file is a hyphen, bdiff reads the standard input. In this case, *file1* cannot be a hyphen.

lines-per-segment

Specifies an optional integer value that bdiff uses as the number of lines into which it divides each segment. If you do not use the *lines-per-segment* argument, bdiff uses a default of 3500. This argument is useful for those cases in which segments of the default size are too large and cause diff to fail. If you use this argument, it must appear after the file arguments. If you also use the -s option, the *lines-per-segment* argument must appear before -s.

-s Suppresses bdiff diagnostic messages. This option does not suppress diff status messages. If you use this option, it must appear after the file arguments. If you also use the *lines-per-segment* argument, the -s option must appear after the *lines-per-segment* argument.

DESCRIPTION

bdiff finds the differences between files that are too large for diff by dividing the files into segments and by then running diff on the corresponding segments.

The output of bdiff is exactly that of diff, with line numbers adjusted to make it appear as though the files were processed as a whole. Note that because bdiff divides the files into segments, it does not necessarily find a smallest sufficient set of file differences.

The output consists of line number information and changed lines in a format that can be used by ed to change *file1* into *file2*. A less than symbol (<) at the beginning of a line indicates that a difference has been found in *file1*. A greater than symbol (>) at the beginning of a line indicates a difference in *file2*. See diff(1) for a complete explanation of the output.

EXAMPLES

Here are the contents of two files named `Harold` and `Maude`:

```
Harold:              Maude:
The first line.      The first line.
The second line.     The second line.
The third line.      The THIRD line.
```

This command line produces the differences between the files:

```
bdiff Harold Maude
```

Here is the output:

```
3c3
< The third line.
---
> The THIRD line.
```

STATUS MESSAGES AND VALUES

The `bdiff` command produces messages that the `help` command can interpret. You may, for example, see this message:

```
bdiff: can not write to temporary file (bd7)
```

To see an explanation of this message, enter

```
help bd7
```

The `bdiff` command can also produce these messages:

```
ERROR: arg count (bd1)
ERROR: both files standard input (bd2)
ERROR: cannot fork, try again (bd3)
ERROR: non-numeric limit (bd4)
ERROR: cannot execute diff (bd5)
```

FILES

`/usr/bin/bdiff`
: Executable file

`/usr/bin/diff`
: Executable file that `bdiff` calls

SEE ALSO

`diff`(1), `diff3`(1), `ed`(1), `help`(1), `sdiff`(1)

NAME

bfs — edits big files

SYNOPSIS

bfs [-] *file*

ARGUMENTS

- Suppresses the display of the size of the file when bfs starts and when you use the w command.

file Specifies the name of the file to edit. This argument is required.

DESCRIPTION

bfs is a read-only editor that can process much larger files than standard editors. Files can contain up to 1 million bytes, with up to 32,000 lines. A line can contain up to 512 characters, including a terminating newline character. If bfs encounters a line that is too long, it displays the message line too long and exits.

The bfs command is similar to ed but more efficient because it does not copy the file to a buffer. You can use bfs to identify and split sections of a large file into small files that can be modified later by a text editor such as vi, or you can use the information gained from using bfs to run csplit on the file.

Prompting

If you type the letter P and press RETURN, bfs displays an asterisk (*) as a prompt, as in ed. You can turn the prompt off by typing P and pressing RETURN again.

Compatibilities with ed

The bfs command supports all address expressions described in ed(1). In addition to the slash (/) and the question mark (?) provided by ed, you can surround regular expressions with the greater-than symbol (>), which indicates downward search without wrap-around, and the lesser-than symbol (<), which indicates upward search without wrap-around. There is a slight difference from ed with regard to mark names: bfs supports only the letters a through z and remembers all 26 marks.

The e, g, v, k, P, p, q, w, =, !, and null commands operate as described in ed(1). The w command works independently from the bfs-specific xo, xt, and xc commands. The bfs command accepts such complex commands as ---, +++-, +++=, -12, and +4p. Note that both 1,10p and 1,10 cause the display of the first ten lines of the file.

The bfs and ed commands both have an f command, but when you are using bfs, the f command only displays the name of the file currently being read; you cannot use it to edit a new file.

bfs-specific Commands

Once you have started bfs on a file, you can use these bfs commands:

:*label*

Creates a label in a command file. If you use optional white-space characters to separate the colon (:) and the label, bfs ignores them. Because bfs does not require that labels be referenced, you can use this command to insert comments in a command file.

(.,.)xb/*regular-expression***/***label*

Jumps upward or downward to *label* if the command succeeds. The xb command is valid only if read from someplace other than a terminal. If the command is read from a pipe, only a downward jump is possible. The command fails under any of these conditions:

- Either address is not between 1 and $.
- The second address is less than the first.
- The regular expression does not match at least one line in the specified range, including the first and last lines.

On success, a period (.) is set to the matched line and a jump is made to *label*. This command is the only command that does not issue an error message on bad addresses, so you can use it to test whether addresses are bad before executing other commands. This xb command is an unconditional jump:

```
xb/^/ label
```

xc [*switch*]

Converts sequences of tabs and spaces to a single space and suppresses blank lines in the output from the p and null commands if the value of *switch* is 1. If the value of *switch* is 0, the action of xc is turned off. If you do not supply *switch*, the action of xc is toggled. When you first start bfs, the action of xc is turned off.

xbz *label*
xbn *label*

Test the exit code of the most recently executed shell command (!*command*) for a zero or a nonzero value, respectively, and go to the specified label if the test succeeds. These two examples search for the next five lines containing the string size:

```
xv55
: here
/size/
xv5!expr %5 - 1
!if [ %5 != 0 ]; then exit 2; fi
```

```
xbn here

xv45
: there
/size/
xv4!expr %4 - 1
!if [ %4 = 0]; then exit 2; fi
xbz there
```

`xf` *command-file*
Specifies a command file from which further `bfs` commands are taken. When `bfs` reaches the end of the file or receives an interrupt signal, or if an error occurs, reading resumes with the file containing the `xf` command. You can nest `xf` commands to a depth of 10.

`xn` Lists the marks currently in use. See the `k` command in `ed`(1) for information about setting marks.

`xo` [*file*]
Diverts further output from the `p` and null commands to *file*, which, if necessary, is created with permission bits of 666. If the file already exists, it is truncated. If you do not specify *file*, `bfs` diverts the output to the standard output.

`xt` *number*
Truncates output from the `p` and null commands to most *number* characters. The default is 255.

`xv` Assigns *value* to the variable specified by *digit*. For example, `xv5100` or `xv5 100` both assign the value `100` to the variable `5`. The command `xv61,100p` assigns the value `1,100p` to the variable `6`. To reference a variable, put a percent sign (`%`) in front of the variable name. For example, using the assignments just given for variables `5` and `6`, these commands cause `bfs` to display the first 100 lines of the file:

```
1,%5p
1,%5
%6
```

Using the assignment just given for variable `5`, this command searches the file for the character string `100` and displays each line that contains a match:

```
g/%5/p
```

To escape the special meaning of `%`, a backslash (`\`) must precede `%`. For example, this command displays the lines that contain `printf` conversion formats for characters, decimal integers, and strings:

```
g/".*\%[cds]/p
```

You can also use the `xv` command to store into a variable the first line of output from a command run by the shell. The only requirement is that the first character of *value* be an exclamation mark (`!`). For example, this sequence stores the first line of the output of `date` into variable `7` and displays its contents:

```
xv7!date
!echo %7
```

This sequence writes the current line to a file called `junk`, stores the contents of `junk` in variable `5`, removes `junk`, and displays the contents of variable `5`:

```
.w junk
xv5!cat junk
!rm junk
!echo "%5"
```

To escape the special meaning of `!` as the first character of *value*, precede it with a backslash (`\`).

EXAMPLES

This command runs `bfs` on a file named `text`:

```
bfs text
```

STATUS MESSAGES AND VALUES

If the `*` prompt is turned on, the `bfs` command displays a question mark (`?`) for errors in commands and displays self-explanatory error messages.

FILES

`/bin/bfs`
Executable file

SEE ALSO

`csplit`(1), `ed`(1)

`regcmp`(3X) in *A/UX Programmer's Reference*

NAME

biff — enables and disables notification of mail by comsat

SYNOPSIS

biff [*switch*]

ARGUMENTS

switch

Sets the state of biff. If *switch* is y, mail notification is enabled. If *switch* is n, mail notification is disabled. If you do not use a *switch* argument, biff displays:

```
is y
```

for enabled, and:

```
is n
```

for disabled.

DESCRIPTION

biff changes the permission bits of your terminal device to enable or disable mail notification by the comsat server. When the execution bit is set for the owner of the terminal device, mail notification is enabled. When the execution bit is not set, comsat does not notify you when new mail arrives.

The best place to include a command to enable biff is in your .login file so that the command is executed each time you log in.

When mail arrives, comsat displays the From:, Subject:, To:, and Date: lines from the mail header and the first few lines of the message on your screen.

For other mail notification methods, see the mail-related environment variables supported by sh, csh, and ksh.

FILES

/usr/ucb/biff

Executable file

SEE ALSO

csh(1), ksh(1), mail(1), sh(1)

comsat(1M) in *A/UX System Administrator's Reference*

NAME

bs — compiles and interprets bs programs

SYNOPSIS

bs [*file* [*argument*]...]

ARGUMENTS

argument
: Specifies an optional argument that bs passes to the program when it executes.

file
: Specifies the name of a source file that bs uses as input before reading from the console. By default, bs compiles statements read from *file* for later execution and immediately executes statements entered from the console. See the compile and execute statements for details.

DESCRIPTION

bs compiles and interprets programs that are written in a language that is a remote descendant of BASIC, SNOBOL4, and C. The language is designed for programming tasks where program development time is as important as the resulting speed of execution. The language minimizes the formalities of data declaration and file manipulation. Line-at-a-time debugging, the trace and dump keywords, and useful run-time error messages simplify program testing. Furthermore, you can debug incomplete programs, test inner functions before outer functions have been written, and test outer functions before inner functions have been written.

Syntax

The bs command accepts programs that are made up of input lines. If the last character on a line is a backslash (\), bs interprets the next line as a continuation of the previous line. Lines can be of this form:

statement
label statement

A *label* is a *name* followed by a colon (:). (For a definition of *name*, see "Expression Syntax" later in this section.) A label and a variable can have the same name.

Statement syntax. A bs statement is either an expression or a keyword followed by zero or more expressions. An expression assigns a value or makes a function call. For details, see "Expression Syntax" later in this section. The keywords clear, compile, !, execute, include, ibase, obase, and run are always executed as they are compiled. Here are the possible keywords:

! *shell-command*
: Causes an immediate escape to the shell to execute *shell-command.*

#*string*
Specifies a comment, which bs ignores.

break
Exits from the innermost for or while loop.

clear
Clears the symbol table and compiled statements immediately.

compile[*expression*]
Executes a clear and compiles succeeding statements immediately (overriding the immediate execution default). The bs command evaluates the optional expression and uses the result as a filename for further input.

continue
Transfers to the loop-continuation of the current for or while loop.

dump[*name*]
Causes bs to display the name and current value of every nonlocal variable. If you provide an optional *name*, bs displays only the specified variable. After an error or interrupt, bs displays the number of the last statement and the user-function trace if turned on.

execute
Changes bs to immediate-execution mode. An interrupt has the same effect. This keyword does not cause internally stored statements to execute. See run for a keyword that does.

exit[*expression*]
Exits bs. If you provide an optional expression, bs evaluates *expression* and uses the result as the exit code.

for *name=expression expression statement*
for *name=expression expression*
...
next
for *expression , expression , expression statement*
for *expression , expression , expression*
...
next
Executes repetitively a statement (first form) or a group of statements (second form) under control of a named variable. The variable takes on the value of the first expression, then is incremented by 1 on each loop, not to exceed the value of the second expression. The third and fourth forms require three expressions separated by commas. The first expression is the initialization, the second is the test (TRUE to continue), and the third is the loop-continuation action, which is commonly an increment.

`freturn`
Signals the failure of a user-written function. For details, see the interrogation operator (`?`) in ''Expression Syntax'' later in this section. If interrogation is not active, `freturn` merely returns 0. If interrogation is active, `freturn` transfers execution to that expression, which may bypass intermediate function returns.

`fun` *name* (*[arguments, ...])[variables, ...]*
...
`nuf`
Defines a function name, arguments, and local variables for a user-written function. Up to ten arguments and local variables are allowed. Such names cannot be arrays, nor can they be associated with I/O. You cannot nest function definitions.

`goto` *name*
Passes control to the internally stored statement with the matching label specified by *name*.

`ibase` *N*
Sets the input base to *N*. The supported values for *N* are 8, 10 (the default), and 16. You can enter the hexadecimal values for 10-15 as `a-f`. To enter a number such as `f0a`, use a leading 0, as in `0f0a`. The `bs` command executes `ibase` immediately.

`if` *expression statement*
`if` *expression...*
[`else`
...]
`fi` Executes if the expression evaluates to a nonzero value. The strings `0` and `""` (null) evaluate to 0. In the second form, an optional `else` allows for a group of statements to be executed when the first group is not. The only keyword permitted on the same line with `else` is `if`; the only keywords permitted on the same line with `fi` are other `fi` keywords. The `bs` command also supports `elif`. Only a single `fi` is required to close an `if...elif...[else...]` sequence.

`include` *expression*
Includes the file specified by *expression*. The file must contain `bs` source statements. Such statements become part of the program being compiled. You cannot nest `include` keywords.

`obase` *N*
Sets the output base to *N*. The supported values for *N* are `8`, `10` (the default), and `16`. You can enter the hexadecimal values for 10-15 as `a-f`. To enter a number such as `f0a`, use a leading 0, as in `0f0a`. The `bs` command executes `obase` immediately.

`onintr` *label*
`onintr`
Provides program control of interrupts. In the first form, control passes to the specified label, just as if a `goto` had been executed at the time `onintr` executed. The effect of `onintr` is cleared after each interrupt. In the second form, an interrupt causes `bs` to terminate.

`return` [*expression*]
Returns from a function call. If present, the optional expression is evaluated and the result is passed back as the value of a function call. If no expression is given, the function call returns 0.

`run`
Resets the random-number generator. Control is passed to the first internally stored statement. If `run` is in a file, it should be the last statement in the file.

`stop`
Stops execution of internally stored statements and causes `bs` to revert to immediate mode.

`trace` [*expression*]
Controls function tracing. If the optional expression is null or evaluates to 0, tracing is turned off. Otherwise, `bs` displays a record of user-function calls and returns. Each `return` decrements the value of *expression*.

`while` *expression statement*
`while` *expression*
`...`
`next`
Executes repetitively a group of statements. The `while` keyword is similar to `for` except that only the conditional expression for loop-continuation is given.

Expression syntax. Unless the final operation is an assignment, `bs` displays the result of an immediate expression statement. Here is the syntax for expressions:

`?` *expression*
Tests for the success of the expression, rather than its value. The interrogation operator is useful for testing for end-of-file condition (as shown in the ''Examples'' section later in this manual page), checking the result of the `eval` built-in function, and checking the return from user-written functions (as described in the discussion of the `freturn` keyword earlier in this section). Execution of an interrogation trap causes an immediate transfer to the most recent interrogation, possibly

skipping assignment statements or intervening function levels.

- *expression*
  Results in the negation of the expression.

--*name*
  Decrements the value of *name*, which is a variable name or an array reference.

++*name*
  Increments the value of *name*, which is a variable name or an array reference.

! *expression*
  Results in the logical negation of the expression.

name
  Specifies a variable. A name begins with a lowercase or uppercase letter, optionally followed by letters and digits. Only the first six characters of a name are significant. Except for names declared in `fun` statements, all names are global to the program. Names can take on numeric (double-float) values or string values, or can be associated with input and output. For details, see the `open` function in ''File-handling Functions'' later in this section.

name(*[expression[,expression]...]*)
  Calls a function. Except for built-in functions, which are listed later in this section, *name* must be defined with a `fun` statement. The `bs` command passes arguments to functions by value.

name [*expression*[,*expression*]...]
  References arrays or tables. See ''Built-in Table Functions'' later in this section for details. For arrays, each expression is truncated to an integer and used as a specifier for the name. The resulting array reference is syntactically identical to a name; `a[1,2]` is the same as `a[1][2]`. The truncated expressions are restricted to values between 0 and 32767.

number
  Represents a constant value. A number is written in Fortran style and contains digits, an optional decimal point, and possibly a scale factor consisting of an `e` followed by an optionally signed exponent.

string
  Specifies a character string that is delimited by a double quotation mark (`"`). The backslash (`\`) is an escape character that allows the double quotation mark (`\"`), newline (`\n`), carriage return (`\r`), backspace (`\b`), and tab (`\t`) characters to appear in a string. Otherwise, the backslash stands for itself.

(*expression*)
Alters the normal order of evaluation. See ''Binary Operators'' later in this section for the normal order of evaluation.

(*expression*, *expression*[, *expression* . . .])[*expression*]
Selects, using a bracketed expression as a subscript, a comma-separated expression from the parenthesized list. List elements are numbered from the left, starting at 0. The expression has the value TRUE if a equals b:

```
(FALSE,TRUE)[a == b]
```

expression operator expression
Abbreviates the common functions of two arguments by separating the two arguments with an operator denoting the function. Except for the assignment, concatenation, and relational operators, both operands are converted to numeric form before the function is applied.

Binary operators. The binary operators are listed in order of increasing precedence, as follows:

= Makes assignments. The left operand must be a name or an array element. The result is the right operand. Assignment binds right to left, whereas all other operators bind left to right.

_ Concatenates. The operator is the underscore character (_).

& | Perform logical operations. The result of the logical AND character (&) is 0 if either of its arguments is 0. The result is 1 if both of its arguments are nonzero. The result of the logical OR character (|) is 0 if both of its arguments are 0. The result is 1 if either of its arguments is nonzero. Both operators treat a null string as 0.

< <= > >= == !=
Perform relational operations. The relational operators are < (less than), <= (less than or equal), > (greater than), >= (greater than or equal), == (equal), and != (not equal). They return 1 if their arguments are in the specified relation. Otherwise, they return 0. The comparison *a*>*b*>*c* is the same as the comparison *a*>*b*&*b*>*c*. If both operands are strings, bs makes a string comparison.

+ -
Perform addition and subtraction.

* / %
Perform multiplication, division, and remaindering.

^ Performs exponentiation.

Built-in Functions

The bs command provides the following built-in functions.

Dealing with arguments. These built-in functions manipulate arguments:

arg(*i*)
: Specifies the value of the *i*th actual parameter on the current level of function call. At level 0, arg returns the *i*th command-line argument. For example, arg(0) returns bs.

narg( )
: Returns the number of arguments passed. At level 0, bs returns the number of command-line arguments.

Mathematical functions. The bs command provides these built-it mathematical functions:

abs(*x*)
: Returns the absolute value of *x*.

atan(*x*)
: Returns the arctangent of *x*. Its value is between -π/2 and π/2.

ceil(*x*)
: Returns the smallest integer not less than *x*.

cos(*x*)
: Returns in radians the cosine of *x*.

exp(*x*)
: Returns the exponential function of *x*.

floor(*x*)
: Returns the largest integer not greater than *x*.

log(*x*)
: Returns the natural logarithm of *x*.

rand()
: Returns a uniformly distributed random number between 0 and 1.

sin(*x*)
: Returns in radians the sine of *x*.

sqrt(*x*)
: Returns the square root of *x*.

String functions. The bs command provides these built-in string functions:

format(*f*, *a*)
: Returns the formatted value of *a*, where *f* is a format specification in the style of printf. The value of *f* can be f, e, or s only.

`index(`*x*`, `*y*`)`
: Returns the number of the first position in *x* that any of the characters from *y* matches. If no match is found, `bs` returns 0.

`match(`*string*`, `*pattern*`)`
`mstring(`*n*`)`
: Match strings. The *pattern* is similar to the regular expression syntax of `ed`. The characters `.`, `[`, `]`, `*`, and `$` are special. The `mstring` function returns the *n*th (1 <= *n* <= 10) substring of the subject that occurred between pairs of the pattern symbols `\(` and `\)` for the most recent call to `match`. To succeed, patterns must match the beginning of the string, as if all patterns begin with `*`. The function returns the number of characters matched. Here is an example:

```
match("a123ab123", ".*\([a-z]\)") == 6
mstring(1) == "b"
```

`size(`*s*`)`
: Returns the size (length in bytes) of *s*.

`substr(`*s*`, `*start*`, `*width*`)`
: Returns the substring of *s* defined by the *start* position and *width*.

`trans(`*s*`, `*f*`, `*t*`)`
: Translates characters of the source *s* from matching characters in *f* to a character in the same position in *t*. Source characters that do not appear in *f* are copied to the result. If the string *f* is longer than *t*, source characters that match in the excess portion of *f* do not appear in the result.

File-handling functions. The `bs` command provides these built-in file handling functions:

`open(`*name*`, `*file*`, `*function*`)`
`close(`*name*`)`
: Perform open and close operations. The *name* argument must be a `bs` variable name passed as a string. For `open`, the *file* argument can be either (1) a `0` (zero), `1`, or `2`, representing standard input, output, or error output, respectively; (2) a string representing a filename; or (3) a string beginning with an exclamation point (`!`) representing a command to be executed by means of `sh -c`. The *function* argument must be either `r` (read), `w` (write), `W` (write without newline), or `a` (append). After a `close` command, `bs` treats *name* as an ordinary variable. The `bs` command makes these calls on startup:

```
open("get", 0, "r")
open("put", 1, "w")
open("puterr", 2, "w")
```

See the ''Examples'' section later in this manual page, for sample code.

`access(`*s*`, `*m*`)`
Executes an `access`(2) system call.

`ftype(`*s*`)`
Returns a single character that indicates the file type: `f` for regular file, `p` for FIFO (that is, named pipe), `d` for directory, `b` for block special, or `c` for character special.

Table-handling functions. The `bs` command provides these built-in table handling functions:

`table(`*name*`, `*size*`)`
Defines a table that is an associatively accessed, single-dimension array. Subscripts (called ''keys'') are strings. Numbers are converted. The *name* argument must be a `bs` variable name, passed as a string. The *size* argument sets the minimum number of elements to be allocated. If a table overflow occurs, `bs` displays an error message and stops.

`item(`*name*`, `*i*`)`
`key()`
Access table elements. The `item` function accesses table elements sequentially. (In normal use, there is no orderly progression of key values.) Where the `item` function accesses values, the `key` function accesses the subscript of the previous `item` call. The *name* argument should not be quoted. Because exact table sizes are not defined, the interrogation operator should be used to detect end-of-table. Here is an example:

```
table("t", 100)
 . . .
# If word contains the string "party",
# the following expression adds one
# to the count of that word:
++t[word]
 . . .
# To print out the key/value pairs:
for i = 0, ?(s = item (t , i)), ++i
if key() put = key()_:_s
```

`iskey(`*name*`, `*word*`)`
Tests whether the key *word* exists in the table *name* and returns 1 for `TRUE` and 0 for `FALSE`.

Miscellaneous functions. The `bs` command provides these miscellaneous built-in functions:

`eval(`*s*`)`
Evaluates the string argument as an expression. The `eval` function is handy for converting numeric strings to numeric internal form. You can also use `eval` as a crude form of indirection, as in this example, which increments the variable `xyz`:

```
name = "xyz"
eval("++"_ name)
```

In addition, `eval` preceded by the interrogation operator permits the user to control `bs` error conditions. Here is an example that returns 0 if `XXX` does not exist, instead of halting:

```
?eval("open(\"X\", \"XXX\", \"r\")")
```

This example executes a `goto` to the label *L*:

```
label="L"
if !(?eval("goto "_ label)) puterr = "no label"
```

`last()`
Returns the most recently computed value when in immediate mode.

`plot(`*request, args*`)`
Produces output on devices recognized by `tplot`(1G). The value of *request* can be from 0 to 12. Calls to the `plot` function use these formats:

`plot(0, `*term*`)`
Causes further `plot` output to be piped into `tplot`(1G) with an argument of `-T`*term*.

`plot(1)`
Erases the plotter.

`plot(2, `*string*`)`
Labels the current point with *string*.

`plot(3,`*x1*`,`*y1*`,`*x2*`,`*y2*`)`
Draws the line between (*x1*,*y1*) and (*x2*,*y2*).

`plot(4,`*x*`,`*y*`,`*r*`)`
Draws a circle with center (*x*,*y*) and radius *r*.

`plot(5,`*x1*`,`*y1*`,`*x2*`,`*y2*`,`*x3*`,`*y3*`)`
Draws an arc (counterclockwise) with center (*x1*,*y1*) and endpoints (*x2*,*y2*) and (*x3*,*y3*).

`plot(6)`
Not implemented.

`plot(7,`*x*`,`*y*`)`
Makes the current point (*x*,*y*).

`plot(8,`*x*`,`*y*`)`
Draws a line from the current point to (*x*,*y*).

`plot(9,`*x*`,`*y*`)`
Draws a point at (*x*,*y*).

`plot(10,`*string*`)`
Sets the line mode to *string*.

`plot(11,`*x1*`,`*y1*`,`*x2*`,`*y2*`)`
Makes (*x1*,*y1*) the lower-left corner of the plotting area and (*x2*,*y2*) the upper right corner of the plotting area.

`plot(12,`*x1*`,`*y1*`,`*x2*`,`*y2*`)`
Causes subsequent *x*(*y*) coordinates to be multiplied by *x1* (*y1*) and then added to *x2* (*y2*) before they are plotted. The initial scaling is `plot(12, 1.0, 1.0, 0.0, 0.0)`.

Some requests do not apply to all plotters. All requests except `0` and `12` are implemented by the piping of characters to `tplot`(1G). See `plot`(4) for more details.

EXAMPLES

This example uses `bs` as a calculator:

```
$ bs
# Distance (inches) light travels in a nanosecond.
186000 * 5280 * 12 / 1e9
11.78496
...

# Compound interest
# (6% for 5 years on $1,000).
int = .06 / 4
bal = 1000
for i = 1 5*4 bal = bal + bal*int
bal - 1000
346.855007
...
exit
```

This example is the outline of a typical `bs` program:

```
# initialize things:
var1 = 1
open("read", "infile", "r")
```

```
...
# compute:
while ?(str = read)
      ...
next
# clean up:
close("read")
...
# last statement executed (exit or stop):
exit
# last input line:
run
```

This example demonstrates I/O:

```
# Copy "oldfile" to "newfile".
open("read", "oldfile", "r")
open("write", "newfile", "w")
...
while ?(write = read)
...
# close "read" and "write":
close("read")
close("write")

# Pipe between commands.
open("ls", "!ls *", "r")
open("pr", "!pr -2 -h 'List'", "w")
while ?(pr = ls) ...
...
# be sure to close (wait for) these:
close("ls")
close("pr")
```

This command line shows a way of running `bs`:

```
bs program 1 2 3
```

The example compiles and executes the file named `program` as well as any statements typed from standard input. The arguments `1`, `2`, and `3` are passed as arguments to the program when it executes.

FILES

`/bin/bs`
: Executable file

SEE ALSO

ed(1), ksh(1), sh(1), tplot(1G)

access(2), intro(3), printf(3S), plot(4) in *A/UX Programmer's Reference*

NAME

cal — displays a calendar

SYNOPSIS

cal [[*month*] *year*]

ARGUMENTS

month

Specifies the month for which cal is to display a calendar. The value of *month* is a number between 1 and 12.

year

Specifies the year for which cal is to display a calendar. The value of *year* can be between 1 and 9999. The cal command adjusts its output to reflect the calendar (Julian or Gregorian) that was in effect for the specified year.

DESCRIPTION

cal displays a calendar. If you do not specify any arguments, cal displays a calendar for the current month.

EXAMPLES

The following command displays a calendar for September 1752, which represents a period of time when the calendar was different than what we currently use:

```
cal 9 1752
```

This command displays a calendar for the year 92 A.D., not the year 1992:

```
cal 92
```

FILES

/usr/bin/cal

Executable file

NAME

calendar — provides a reminder service

SYNOPSIS

calendar [-]

ARGUMENTS

- Causes calendar to examine the calendar file in the login directory of each user on the system and send the output of calendar as a mail message. The system administrator can also put the

  ```
  calendar -
  ```

  command in the /usr/spool/crontabs file so that cron can run it automatically.

DESCRIPTION

calendar examines the file calendar in the current directory and displays the lines that contain today's or tomorrow's date anywhere in the line. On a Friday, calendar extends "tomorrow" through Monday. An easy way to use calendar is to keep your calendar file in your login directory and to run calendar from your .login file.

The calendar command understands dates such as Mar. 7, march 7, and 3/7, but it does not understand 7 March.

EXAMPLES

Suppose that your calendar file contains the lines

```
5/3   Bruce's birthday
Monday, September 6     Labor Day Holiday
Mar 14      Status report due.
```

and the date is March 13 or 14. The command:

```
calendar
```

displays this reminder:

```
Mar 14      Status report due.
```

LIMITATIONS

The calendar command does not take holidays into account when determining what "tomorrow" is.

NOTES

The calendar command requires, at minimum, two steps: (1) that you use a text editor to enter information in your calendar file and (2) that you remember to run the calendar command. You may want to use the at command because it provides a reminder service that combines the entry of reminder information (including a specific time to be reminded)

with the execution of a single command.

Another reminder service is provided by the `leave` command, which persistently reminds you when you need to log out.

FILES

`./calendar`
: File that contains reminders

`/etc/passwd`
: File that `calendar` examines to get login directories

`/tmp/cal*`
: Temporary files

`/usr/bin/calendar`
: Executable file

`/usr/lib/calprog`
: Executable file that produces a temporary file containing today's and tomorrow's date in a format suitable for processing by `calendar`

SEE ALSO

`at`(1), `crontab`(1), `leave`(1), `mail`(1)

`cron`(1M) in *A/UX System Administrator's Reference*

NAME

cancel — cancels print requests spooled through the lp command

SYNOPSIS

cancel [*printer*]

cancel [*id*]...

ARGUMENTS

id Specifies the request identification number that you want to delete. The *id* is part of the message returned by lp when you spool a print job, such as laser-1233.

printer

Specifies a particular printer that is currently printing the job you want to stop. To display a complete list of printers, use lpstat(1).

DESCRIPTION

cancel deletes jobs from the print queue. Jobs are placed in the print queue by means of lp (see lp(1)).

When you cancel a request that is currently printing, you free the printer to print its next available request.

FILES

/usr/bin/cancel

Executable file

SEE ALSO

enable(1), lp(1), lpq(1), lpr(1), lprm(1), lpstat(1)

accept(1M), lpadmin(1M), lpsched(1M), reject(1M) in *A/UX System Administrator's Reference*

A/UX Local System Administration

NAME

cat — catenates and displays the contents of files

SYNOPSIS

cat [-] [-e] [-s] [-t] [-u] [-v [*file*]...

ARGUMENTS

- `-` Causes `cat` to read from the standard input.
- `-e` Causes `cat` to display a dollar sign (`$`) before the newline character at the end of each line. This option must be used in conjunction with the `-v` option; if it is not, `cat` ignores this option.
- *file* Specifies a file for `cat` to read. If you do not use any *file* arguments, `cat` reads from the standard input.
- `-s` Causes `cat` to be silent if a specified file does not exist.
- `-t` Causes the tab character to be displayed as `^I` and the form feed character to be displayed as `^L`. This option must be used in conjunction with the `-v` option; if it is not, `cat` ignores this option.
- `-u` Causes `cat` to not buffer the output.
- `-v` Causes `cat` to display nonprinting characters (with the exception of the form feed, newline, and tab characters) in a special way. If you use this option, `cat` displays control characters as `^X` (CONTROL-X) and the delete character (octal 0177) as `^?`. Non-ASCII characters (characters whose high-order bit is set) are displayed as `M-`*x*, where *x* is the character specified by the seven low-order bits.

DESCRIPTION

`cat` reads one or more files and writes the contents of each file to the standard output.

EXAMPLES

This command displays the file named `rose`:

```
cat rose
```

This command catenates the contents of `rose` and `wisteria` and places the result in `flowers`:

```
cat rose wisteria > flowers
```

WARNINGS

This command causes the data in `file1` to be lost:

```
cat file1 file2 > file1
```

Therefore, take care when using shell metacharacters to perform I/O redirection.

FILES

/bin/cat
Executable file

SEE ALSO

cp(1), head(1), more(1), pg(1), pr(1), tail(1)

NAME

cb — improves spacing and indentation of C source files

SYNOPSIS

cb [-j] [-l *line-length*] [*file*]...

cb [-j] [-s] [*file*]...

ARGUMENTS

file Specifies the name of a C source file. If you do not specify a *file* argument, cb reads from the standard input.

-j Causes split lines, such as those split by the -l option, to be put back together.

-l *line-length*
Causes cb to attempt to split lines that are longer than the value specified by *line-length*. Not all lines are split; for example, cb does not split comments or lines that contain long stretches of characters surrounded by quotation marks. If you do not use this option, cb preserves the position of newline characters in the C source file.

-s Causes cb to produce an output that is in the canonical style defined by Kernighan and Ritchie in *The C Programming Language*.

DESCRIPTION

cb reads C files and writes them to the standard output with spacing and indentation that displays the structure of the code.

A more recent program called indent performs the same function as this command. For details, refer to the indent(1) manual page.

EXAMPLES

A sample C source file contains these lines:

```
#define COMING 1
#define GOING 0

main ()
{
/*  This is a test of the C Beautifier  */
if (COMING)
printf ("Hello, world\n");
else
printf ("Good bye, world\n");
}
```

Running the cb command on the sample C source file produces this output:

```
#define COMING 1
#define GOING 0
```

```
main ()
{
        /*  This is a test of the C Beautifier  */
        if (COMING)
                printf ("Hello, world\n");
        else
                printf ("Goodbye, world\n");
}
```

LIMITATIONS

Punctuation in preprocessor statements causes indentation errors.

FILES

`/usr/bin/cb`
Executable file

SEE ALSO

`cc`(1), `indent`(1)

The C Programming Language by B. W. Kernighan and D. M. Ritchie (New Jersey: Prentice-Hall, Inc., 1978)

NAME

cc — invokes the C compiler

SYNOPSIS

cc [-A *factor*] [-a] [-B *string*] [-c] [-C] [-D*symbol*[=*def*]] [-E] [-fm68881] [-F] [-g] [-I*dir*] [-l*x*] [-L *dir*] [-n] [-o *outfile*] [-O] [-p] [-P] [-R] [-s] [-S] [-t [*p012al*]] [-T] [-U*symbol*] [-v] [-W *c*,*arg1*[,*arg2*]... [-X] [-y] [-Z*flags*] [-68030] [-68040] [-68851] [-#]... *file*...

ARGUMENTS

-# Echoes the names and arguments of subprocesses that would have started, without actually starting the program. This is a special debug option.

-A *factor*
Expands the default symbol table allocations for the compiler, assembler, and link editor. The default allocation is multiplied by the *factor* given.

-a Includes source code as comments in the assembly file generated with the -S option.

-B *string*
Constructs pathnames for substitute preprocessor, compiler, assembler, and link-editor passes by concatenating *string* with the suffixes cpp, comp, optim, as, and ld. If *string* is empty, it is taken to be /lib/. For example, versions of the C compiler, assembler, and link editor can be found in the directory /usr/lib/big. These tools operate just like their standard counterparts, except that their symbol tables are very large. If you receive an overflow error message when you compile your program with the standard versions, you may wish to switch to the alternate versions using

```
cc -B "/usr/lib/big/" -o filename filename.c
```

You should have 4 MB or more of main memory in order to use the big versions of these programs safely.

-c Suppresses the link-editing phase of the compilation and forces an object file to be produced even if only one program is compiled.

-C Passes along all comments except those found on cpp(1) directive lines. The default strips out all comments.

-D*symbol*[=*def*]
Defines the external *symbol* to the preprocessor and gives it the value *def* (if specified). If no *def* is given, *symbol* is defined as 1. This mechanism is useful for conditional compilation using preprocessor

control lines.

`-E` Runs only `cpp`(1) on the named C programs and sends the result to the standard output.

file Specifies the file that is to be compiled.

`-fm68881`
Generates inline code for the MC68881 floating-point coprocessor. This is the default.

`-F` Does not generate inline code for the MC68881 floating-point coprocessor.

`-g` Generates additional information needed for the use of `sdb`(1).

`-I`*dir*
Searches for `#include` files (whose names do not begin with `/`) in *dir* before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in `" "` (double quotes) are initially searched for in the directory of the `.c` file currently being compiled, then in directories named in `-I` options, and finally in directories on a standard list. For `#include` files whose names are enclosed in <>, the directory of the `.c` file is not searched.

`-l`*x* Searches the library `lib`*x*`.a`, where *x* is up to 7 characters long. A library is searched when its name is encountered, so the placement of `-l` is significant. By default, libraries are located in `LIBDIR`. If you plan to use the `-L` option, that option *must* precede `-l` on the command line. Same as `-l` in `ld`(1).

`-L` *dir*
Changes the algorithm of searching for `lib`*x*`.a` to look in *dir* before looking in `LIBDIR`. This option is effective only if it precedes the `-l` option on the command line. Same as `-L` in `ld`(1).

`-n` Arranges for the loader to produce an executable which is linked in such a manner that the text can be made read-only and shared (nonvirtual) or paged (virtual).

`-o` *outfile*
Produces an output object file, *outfile*. The default name of the object file is `a.out`. Same as `-o` in `ld`(1).

`-O` Invokes an object-code optimizer. The optimizer moves, merges, and deletes code, so symbolic debugging with line numbers could be confusing when the optimizer is used. For this reason, use of the `-g` option disables the `-O` option. This option may not work properly on code containing `asm` directives.

`-p` Arranges for the compiler to produce code that counts the number of times each routine is called. Also, if link-editing takes place, replace the standard startoff routine by one that automatically calls `monitor`(3C) at the start and arranges to write out a `mon.out` file at normal termination of execution of the object program.

`-P` Runs only `cpp`(1) on the named C programs and leaves the result on corresponding files suffixed `.i`.

`-R` Causes the assembler to remove its input file when finished.

`-s` Strips the line-number entries and symbol-table information from the output of the object file. Same as `-s` in `ld`(1).

`-S` Compiles, but does not assemble, the named C programs and leaves the assembly-language output on corresponding files suffixed `.s`.

`-t` [*p012al*]
Finds only the designated preprocessor passes whose names are constructed with the *string* argument of the `-B` option; that is, (`p`), compiler (`0` and `1`), optimizer (`2`), assembler (`a`), and link editor (`l`). In the absence of a `-B` option and its argument, *string* is taken to be `/lib/n`. Using the `-t` option with no argument is equivalent to `-tp012`.

`-T` Truncates symbol names to 8 significant characters. Many modern C compilers, as well as the proposed ANSI standard for C, allow arbitrary-length variable names. `cc` follows this convention. The `-T` option is provided for compatibility with earlier systems.

`-U`*symbol*
Undefines *symbol* to the preprocessor.

`-v` Prints the command line for each subprocess executed.

`-W` *c*, *arg1*[, *arg2*]
Hands off the argument(s) *argi* (where $i = 1,2, \ldots, n$) to pass *c*, where *c* is one of [p012al] indicating preprocessor, compiler first pass, compiler second pass, optimizer, assembler, or link editor, respectively. For example:

```
-Wa,-m
```

invokes the `m4` macro preprocessor on the input to the assembler. (The `-m` option to `as` causes it to go through `m4`.) This must be done for a source file that contains assembler escapes.

`-X` Ignored by A/UX® for the Motorola 68020 and 68030 host processors.

`-y` Suppresses searching of `/usr/include` for header files and instead searches only in directories specified by the `-I` option.

-Z*flags*
Specifies special *flags* to override the default behavior (see NOTES in this section). Currently recognized flags are:

c Return pointers in a0 without copying to d0.

n Emits no code for stack-growth. This is the default.

m Uses Motorola SGS-compatible stack growth code.

p Uses `tst.b` stack probes.

E Ignores all environment variables.

I Emits inline code for the MC68881 floating-point coprocessor. This is the default.

l Suppresses selection of a loader command file.

t Does not delete temporary files.

S Compiles to be SVID-compatible. Links the program with a library module that calls `setcompat`(2) with the `COMPAT_SVID` flag set. Defines only the `SYSV_SOURCE` feature test macro.

P Compiles for the POSIX environment. Links the program with a library module that calls `setcompat`(2) with the `COMPAT_POSIX` flag set. Defines only the `POSIX_SOURCE` feature test macro.

B Compiles to be BSD-compatible. Links the program with a library module that calls `setcompat`(2) with the `COMPAT_BSD` flag set. Defines only the `BSD_SOURCE` feature test macro.

-68030
Directs the assembler to recognize the memory management unit (MMU) instructions for a Motorola 68030 processor.

-68040
Directs the assembler to recognize the instructions for a Motorola 68040 processor.

-68851
Directs the assembler to recognize the coprocessor instructions for a Motorola 68851 PMMU. This is the default.

DESCRIPTION
`cc` is a front-end program that invokes the preprocessor, compiler, assembler, and link editor, as appropriate. The default is to invoke each one in turn.

Arguments whose names end with `.c` are taken to be C source programs. They are compiled, and each object program is left in a file in the current directory, whose name is that of the source, with `.o` substituted for `.c`. In the same way, arguments whose names end with `.s` are taken to be assembly source programs and are assembled to produce a `.o` file. By default, the named files are loaded to produce an output file named `a.out`. If a single C program is compiled and loaded all at once, the `.o` file is deleted.

Other arguments are taken to be link-editor flag-option arguments, C-compatible object programs (typically produced by an earlier run of `cc`), or libraries of C-compatible routines. These programs, together with the results of any compilations specified, are link edited (in the order given) to produce an executable program with the name `a.out` unless the `-o` option of the link editor is used.

WARNINGS

By default, the return value from a C program is completely random. The only two guaranteed ways to return a specific value are to call `exit` explicitly (see `exit`(2)) or to leave the function `main()` with a `return(`*expression*`)` statement.

STATUS MESSAGES AND VALUES

The status messages produced by the C compiler are sometimes cryptic. Occasional messages may be produced by the assembler or link editor.

NOTES

This version of `cc` is based on the `cc` released with the Motorola SGS and has been changed in the following ways:

- The `-Z` option has been added to explicitly control generation of stack-growth code for cross-development environments or generation of stand-alone code. The Motorola SGS looks for an environment variable called `M68000` and generates stack-growth code if the variable is set to `STACKCHECK`. This `cc` defaults to no stack-growth code on the Macintosh II® 68020 and 68030 processors.
- The default is to produce shared text programs. To produce nonshared text programs, you must run `ld` with the `-N` option.
- When `cc` is used with the `-g` option, the arguments `-u _dbargs -lg` are inserted in the command line for the link phase. This causes the contents of `libg.a` to be linked in. Note that the Motorola SGS only generates the loader argument `-lg`, which is not sufficient to cause loading of the library's contents.

- The `-v` (verbose) option has been added to print the command line for each subprocess executed. This helps to isolate problems to a specific phase of the compilation process by showing exactly what `cc` is doing, so that each phase can be run by hand, if necessary.
- The Motorola SGS compiler expects functions that return pointers or structures to return their values in a0 and expects other functions to return their values in d0/d1. Because of the large body of existing code that has inconsistent type declarations, this version of the compiler emits code to return pointers in both a0 and d0 by copying a0 to d0 just prior to returning. This copy operation can be suppressed with the `-Zc` option, thus generating slightly smaller code.

FILES

`/usr/bin/cc`
: Executable file

file`.c`
: Input file

file`.o`
: Object file

file`.s`
: Assembly language file

`a.out`
: Link-edited output file

`/usr/tmp/mc68?`
: Temporary file

`/lib/cpp`
: Preprocessor file

`/lib/comp`
: Compiler file

`/lib/optim`
: Optimizer file

`/bin/as`
: File containing the assembler

`/bin/ld`
: File containing the link editor

`/lib/libc.a`
: Standard library file

`/lib/libposix.a`
: POSIX library file

`/lib/libbsd.a`
: BSD library file

`/lib/libsvid.a`
SVID library file

`/usr/lib/shared.ld`
Loader command file for shared text or paged programs

`/usr/lib/shlib.ld`
Loader command file for shared text or paged programs using shared libraries

`/usr/lib/unshared.ld`
Loader command file for unshared text programs

`/usr/lib/unshlib.ld`
Loader command file for unshared text programs using shared libraries

`/lib/crt0.o`
Run-time startoff file

`/lib/crt1.o`
Run-time startoff file with shared library support

`/lib/crt2.o`
Run-time startoff file used with `crt1.o` for shared library support

`/lib/crtn.o`
Run-time startoff file used with `crt1.o` and `crt2.o` for shared library support

`/lib/mcrt0.o`
Run-time startoff file for profiling

SEE ALSO

`as`(1), `dis`(1), `ld`(1)

`setcompat`(2) in *A/UX Programmer's Reference*

The C Programming Language by B. W. Kernighan and D. M. Ritchie, (New Jersey, Prentice-Hall: 1978)

"`cc` Command Syntax," in *A/UX Programming Languages and Tools, Volume 1*

"A/UX POSIX Environment," in *A/UX Programming Languages and Tools, Volume 1*

See compact(1)

NAME

cdc — changes the delta commentary of an SCCS delta

SYNOPSIS

cdc [-m[*mrlist*]] -r *SID* [-y[*comment*]] *file...*

ARGUMENTS

file Specifies the file to be changed. If a name of - is given, the standard input is read (see WARNINGS); each line of the standard input is taken to be the name of an SCCS file to be processed.

-m[*mrlist*]

Allows you to supply a list of MR numbers to be added and/or deleted in the delta commentary of the SID specified by the -r option, if the SCCS file has the v option set (see admin(1)). A null MR list has no effect.

MR entries are added to the list of MRs in the same manner as that of delta(1). In order to delete an MR, precede the MR number with the character ! (see EXAMPLES). If the MR to be deleted is currently in the list of MRs, it is removed and changed into a comment line. A list of all deleted MRs is placed in the comment section of the delta commentary and preceded by a comment line stating that they were deleted.

If the -m option is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see the -y option).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the v option has a value (see admin(1)), it is taken to be the name of a program (or shell procedure) that validates the correctness of the MR numbers. If a nonzero exit status is returned from the MR number validation program, cdc terminates and the delta commentary remains unchanged.

-r *SID*

Specifies the SCCS Identification (SID) string of a delta for which the delta commentary is to be changed.

-y[*comment*]

Specifies arbitrary text used to replace the *comment*(s) already existing for the delta specified by the -r option. The previous comments are kept and preceded by a comment line stating that they were changed. A null comment has no effect.

If the -y option is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the *comment* text.

DESCRIPTION

cdc changes the ''delta commentary'', for the SID specified by the -r option, of each named SCCS file.

A ''delta commentary'' is defined to be the Modification Request (MR) and comment information normally specified via the delta(1) command (-m and -y options).

If a directory is named, cdc behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

The exact permissions necessary to modify the SCCS file are documented in the ''SCCS Reference'' in *A/UX Programming Languages and Tools, Volume 2*. Simply stated, they are either (1) if you made the delta, you may change its delta commentary; or (2) if you own the file and directory, you may modify the delta commentary.

EXAMPLES

The command:

```
cdc -r1.6 -m"bl78-12345 !bl77-54321 bl79-00001" -ytrouble s.file
```

adds bl78-12345 and bl79-00001 to the MR list, removes bl77-54321 from the MR list, and adds the comment trouble to delta 1.6 of s.file.

The command:

```
cdc -r1.6 s.file
MRs?
!bl77-54321 bl78-12345 bl79-00001
comments?
trouble
```

does the same thing.

STATUS MESSAGES AND VALUES

Use help for explanations.

WARNINGS

If SCCS filenames are supplied to the cdc command via the standard input (- on the command line), then the -m and -y options must also be used.

FILES

`/usr/bin/cdc`
Executable file

SEE ALSO

`admin`(1), `delta`(1), `get`(1), `help`(1), `prs`(1)

`sccsfile`(4) in *A/UX Programmer's Reference*

"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*

NAME

cflow — generates a C flowgraph

SYNOPSIS

cflow [-d*num*] [-i_] [-ix] [-r] *file...*

ARGUMENTS

-d*num*
: Specifies the depth (*num*) at which the flowgraph is cut off. By default this is a very large number. Any attempts to set the cutoff depth to a nonpositive integer will be met with contempt.

file Specifies the file to be analyzed.

-i_
: Includes names that begin with an underscore. The default is to exclude these functions (and data if -ix is used).

-ix
: Includes external and static data symbols. The default is to include only functions in the flowgraph.

-r Reverses the caller:callee relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

DESCRIPTION

cflow analyzes a collection of C, yacc, lex, assembler, and object files and attempts to build a graph charting the external references. Files suffixed in .y, .l, .c, and .i are yacc'd, lex'd, and C-preprocessed (bypassed for .i files) as appropriate and then run through the first pass of lint(1). (The -I, -D, and -U options of the C-preprocessor are also understood.) Files suffixed with .s are assembled and information is extracted (as in .o files) from the symbol table. The output is collected and turned into a graph of external references, which is displayed upon the standard output.

Each line of output begins with a reference (i.e., line) number, followed by a suitable number of tabs indicating the level. Then the name of the global (normally only a function not defined as an external or beginning with an underscore; see below for the -i inclusion option), a colon, and its definition. For information extracted from C source, the definition consists of an abstract type declaration (e.g., char *), and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (e.g., *text*). Leading underscores in C-style external names are deleted.

Once a definition of a name has been printed, subsequent references to that name contain only the reference number of the line where the definition may be found. For undefined references, only < > is printed.

When the nesting level becomes too deep, the `-e` option of `pr`(1) can be used to compress the tab expansion to something less than every eight spaces.

EXAMPLES

Given the following in `file.c`:

```
int    i;

main()
{
        f();
        g();
        f();
}

f()
{
        i = h();
}
```

the command:

```
cflow -ix file.c
```

produces the output:

```
1       main: int(), <file.c 4>
2             f: int(), <file.c 11>
3                   h: <>
4                   i:int, <file.c 1>
5             g: <>
```

STATUS MESSAGES AND VALUES

Complains about bad options. Complains about multiple definitions and only believes the first. Other messages may come from the various programs used (such as the C-preprocessor).

LIMITATIONS

Files produced by `lex`(1) and `yacc`(1) cause the reordering of line number declarations, which can confuse `cflow`. To get proper results, feed `cflow` the `yacc` or `lex` input.

FILES

/usr/bin/cflow
: Executable file

/usr/lib/lpfx
: File that filters line(1) output into dag input

/usr/lib/nmf
: File that converts nm output into dag input

/usr/lib/dag
: File containing a graph maker

/usr/lib/flip
: File containing a reverser

SEE ALSO

as(1), cc(1), cpp(1), lex(1), lint(1), nm(1), pr(1), yacc(1)

NAME

`changesize` — changes or displays the fields of the `SIZE' resource of a file

SYNOPSIS

`changesize` [±*option*] [`-m`*minsize*] [`-p`*prefsize*] [`-v`] *file*

ARGUMENTS

+*option*

-*option*

Sets or clears the MultiFinder flag specified by *option*. +*option* sets the MultiFinder flag; -*option* clears the flag. You can specify multiple options at the same time on the command line. Here are the MultiFinder flags that can be modified:

file Specifies the Macintosh file that you want to change or query. If the resource file is a separate file (prefixed with a percent sign), as in the case of a file in Apple Double format, you should specify the data fork file without the prefix, allowing `changesize` to locate the resource file itself.

`-m`*minsize*

Specifies the minimum RAM required by an application.

`32BitCompatible`

If set, indicates that your application is 32-bit clean.

`CanBackground`

If set, indicates that NULL events can be set while in the background.

`ChildDiedEvents`

If set, indicates that debuggers can set `ChildDiedEvents` to 1. Normally set to 0.

`GetFrontClicks`

Sets this flag if you want to receive the mouse-down and mouse-up events. These events bring your application to the foreground when the user clicks in one of the windows of your application while it is suspended.

`MultiFinderAware`

Responsibile for activating and deactivating any windows in response to a suspend/resume event, if set.

`OnlyBackground`

Sets this flag if your application does not have a user interface and will not run in the foreground.

`OptionSwitch`
Sets Switcher compatibility. Normally set to 1.

`SaveScreen`
Sets Switcher compatibility. Normally set to 0.

`SuspendResume`
Signifies that the application knows how to process suspend/resume events, if set.

`-p`*prefsize*
Specifies an amount of memory in which the application will run effectively and that MultiFinder attempts to secure upon launch of the application. This value is expressed in units of kilobytes (KB).

`-v` Prints the values of fields in the 'SIZE' resource and then exits without changing anything.

DESCRIPTION

`changesize` is based on an MPW tool that prints the fields of the 'SIZE' resource of an application and allows the user to modify any of the fields of the 'SIZE' resource. The 'SIZE' resource contains MultiFinder flags followed by the preferred size and minimum size of the application.

EXAMPLES

To print the fields of the 'SIZE' resource in a file named `SpiffWriter`, you could enter the following command:

```
/mac/bin/changesize -v SpiffWriter
```

To set the `32BitCompatible` flag, clear the `CanBackground` flag, and set the preferred memory size to 500 KB for the same file, you could enter the following command:

```
/mac/bin/changesize +32BitCompatible  \
-CanBackground -p500 SpiffWriter
```

LIMITATIONS

The `changesize` command is not supported in 24-bit mode. You must run it from the command line while logged in with a 32-bit Macintosh session type. (See `Login`(1) for more information regarding session types.)

FILES

`/mac/bin/changesize`
Executable file

SEE ALSO

Login(1), setfile(1).

See cw(1)

See eqn(1)

NAME

checkinstall — checks the installation of boards

SYNOPSIS

checkinstall ethertalk

ARGUMENTS

ethertalk

Indicates that the Apple EtherTalk board should be checked.

DESCRIPTION

checkinstall performs a quick test to see if the named board has been installed or not. The only board type currently supported is the Apple EtherTalk board.

FILES

/etc/checkinstall

Executable file

SEE ALSO

etheraddr(1M) in *A/UX Programmer's Reference*

NAME

checkmm, checkmm1 — check documents formatted with the mm macros

SYNOPSIS

checkmm *file...*

ARGUMENTS

file Specifies the file to be checked.

DESCRIPTION

checkmm stands for "check memorandum macros." Use checkmm to check for syntax errors in files that have been prepared for the mm(1) or mmt(1) command. For example, checkmm checks that you have a .DE (display end macro) corresponding to every .DS (display start macro).

The output for checkmm is the number of lines checked, and a list of macros that are unfinished because of missing macros. If you do not include a filename on the command line, checkmm takes input from standard input.

STATUS MESSAGES AND VALUES

checkmm Cannot open *file*

The *file* is unreadable. The remaining output of the program is diagnostic of the source file.

FILES

/usr/bin/checkmm

Executable file

/usr/bin/checkmm1

Executable file

SEE ALSO

eqn(1), mm(1), mmt(1), mvt(1), neqn(1), tbl(1)

mm(5) in *A/UX Programmer's Reference*

"Other Text Processing Tools" in *A/UX Text Processing Tools*

"mm Reference" in *A/UX Text Processing Tools*

See checkmm(1)

NAME

checknr — checks nroff/troff files

SYNOPSIS

checknr [-a *.x1 .y1 .x2 .y2 .... .xn .yn*] [-c *.x1 .x2 .x3 .... .xn*] [-f] [-s] [*file*]...

ARGUMENTS

[-a *.x1 .y1 .x2 .y2 .... .xn .yn*]

Allows additional pairs of macros to be added to the list. This must be followed by groups of six characters, each group defining a pair of macros. The six characters are a period, the first macro name, another period, and the second macro name. For example, to define a pair .BS and .ES, use:

```
checknr -a.BS.ES
```

[-c *.x1 .x2 .x3 .... .xn*]

Causes commands (macros) to be considered "defined" that would otherwise be complained about as undefined. For instance, user-defined macros are not part of the ms macro package, and thus would be considered undefined. Any macros to be defined for checknr follow the -c with no spaces. For example, to define the macros .XX and .YY, use:

```
checknr -c.XX.YY
```

[-f]

Requests checknr to ignore \f font changes.

[-s]

Requests checknr to ignore \s size changes.

DESCRIPTION

checknr checks a list of nroff(1) or troff(1) input files for certain kinds of errors involving mismatched opening and closing delimiters and unknown commands. If no files are specified, checknr checks the standard input. Delimiters checked are:

(1) Font changes using \f*x*...\fP

(2) Size changes using \s*x*...\s0

(3) Macros that come in *open...close* forms, for example, the .TS and .TE macros, which must always come in pairs.

checknr operates on the ms(5) macro package only.

The checknr command is intended to be used on documents that are prepared with checknr in mind, much the same as lint(1). It expects a certain document writing style for \f and \s commands, in that each \f*x* must be terminated with \fP and each \s*x* must be terminated with \s0.

While it will work to go directly into the next font or to specify the original font or point size explicitly, and many existing documents actually do this, such a practice will produce complaints from `checknr`. Since it is probably better to use the `\fP` and `\s0` forms anyway, you should think of this as a contribution to your document preparation style.

STATUS MESSAGES AND VALUES

Complains about unmatched delimiters.

Complains about unrecognized commands.

Various complaints about the syntax of commands.

LIMITATIONS

There is no way to define a 1-character macro name using the `-a` option.

This command does not recognize certain reasonable constructs correctly, such as conditionals.

FILES

`/usr/ucb/checknr`
Executable file

SEE ALSO

`nroff`(1), `troff`(1)

`ms`(5) in *A/UX Programmer's Reference*

"Other Text Processing Tools" in *A/UX Text Processing Tools*

"`ms` Reference" in *A/UX Text Processing Tools*

NAME

chfn — changes the real-name field of your password file entry for use by finger

SYNOPSIS

chfn [*login-name*]

ARGUMENTS

login-name

Specifies the login name of the user whose entry in the /etc/passwd file is to be changed. Only the system administrator can specify the login name of another user.

DESCRIPTION

chfn changes the real-name field of your entry in the /etc/passwd file. The new information, which consists of your real name, office address, office phone number, and home phone number, is used by finger and other commands. For example, lp prints this information on the banner page.

The chfn command asks you for information by using a prompt that ends with a colon (:). The prompt encloses in brackets the current value, if any. To accept the current value, press RETURN. To enter a blank for a current value, type the word none. You can enter phone numbers with or without hyphens.

When prompted for your office address, you may want to enter your mail stop because the finger command uses the heading Mail Stop to report the office address information.

EXAMPLES

In this example, the user accepts the current value of his name, adds his office address and phone number, and makes blank the value of his home phone number:

```
Name [Biff Studsworth II]:
Office Address (Exs: Grey 222 or MS 32C) []: 521E
Office Phone (Ex: 845-9934 or x-378) []: 1863
Home Phone (Ex: 9875432) [5441546]: none
```

After you run chfn, you should run finger to make sure your entry is the way you want it.

STATUS MESSAGES AND VALUES

Because two users may try to write to /etc/passwd at the same time, commands that modify /etc/passwd make a copy of /etc/passwd and call it /etc/ptmp. If /etc/ptmp already exists because another user is running a command (such as adduser, chsh, passwd, yppasswd, or vipw) that also creates /etc/ptmp, chfn displays this message:

```
Temporary file busy -- try again
```

In this case, try running `chfn` again in a few seconds.

FILES

`/usr/ucb/chfn`
: Executable file

`/etc/passwd`
: File that is modified by `chfn`

`/etc/ptmp`
: File containing a temporary copy of `/etc/passwd`

SEE ALSO

`finger`(1)

`passwd`(4) in *A/UX Programmer's Reference*

See chown(1)

NAME

chmod — changes the permissions of a file

SYNOPSIS

chmod *mode file...*

ARGUMENTS

file Specifies the files that will have its permissions changed.

mode

Specifies the mode to which the file will be changed. The modes are:

4000
: Sets user ID on execution.

2000
: Sets group ID on execution.

1000
: Sets the sticky bit, see chmod(2).

0400
: Allows read access by owner.

0200
: Allows write access by owner.

0100
: Allows execution (search in directory) by owner.

0070
: Allows read, write, and execution by group.

0007
: Allows read, write, and execution by others.

DESCRIPTION

The permissions of the named *files* are changed according to *mode*, which may be absolute or symbolic. An absolute *mode* is an octal number constructed from the R of the modes shown above.

A symbolic *mode* has the form:

[*who*] *op permission* [*op permission*]

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

The *op* option can be + to add *permission* to the file's mode, - to take away *permission*, or = to assign *permission* absolutely (all other bits will be reset).

The *permission* option is any combination of the letters `r` (read), `w` (write), `x` (execute), `s` (set owner or group ID) and `t` (save text, or sticky); `u`, `g`, or `o` indicate that *permission* is to be taken from the current mode. Omitting *permission* is only useful with = to take away all permissions.

Multiple symbolic modes separated by commas may be given. Operations are performed in the order specified. The letter `s` is only useful with `u` or `g` and `t` only works with `u`.

Only the owner of a file (or the superuser) may change its mode. Only the superuser may set the sticky bit. In order to set the group ID, the group of the file must correspond to your current group ID.

EXAMPLES

The command:

`chmod 755` *filename*

changes the mode of `filename` to: read, write, execute (`400+200+100`) by owner; read, execute (`40+10`) for group; read, execute (`4+1`) for others. An `ls -l` of filename shows [`-rwxr-xr-x` *filename*] that the requested mode is in effect.

The command:

`chmod =` *filename*

will take away all permissions from *filename*, including yours. The command:

`chmod o-w` *file*

denies write permission to others. The command:

`chmod +x` *file*

makes a file executable.

FILES

`/bin/chmod`
Executable file

SEE ALSO

`ls`(1), `chown`(1), `csh`(1), `ksh`(1), `sh`(1)

`chmod`(2) in *A/UX Programmer's Reference*

A/UX Essentials

A/UX Local System Administration

NAME

chown, chgrp — change the owner or group of a file

SYNOPSIS

chown *owner file...*

chgrp *group file...*

ARGUMENTS

file Specifies the file to be changed.

group
Specifies the group from which *file* can be accessed.

owner
Specifies the owner of the *file*.

DESCRIPTION

chown changes the owner of the *files* to *owner*. The owner may be either a decimal user ID or a login name found in the password file.

chgrp changes the group ID of the *files* to *group*. The group may be either a decimal group ID or a group name found in the group file.

If either command is invoked by other than the superuser, and the *files* specified are either local or remoted mounted from another System V system, the set-user ID and set-group ID bits of the file mode, 04000 and 02000 respectively, will be cleared.

EXAMPLES

The command:

```
chown doc filea fileb filec
```

would make doc the owner of the three files.

FILES

/bin/chown
Executable file

/bin/chgrp
Executable file

/etc/group
File containing group IDs

/etc/passwd
File containing user IDs

SEE ALSO

chmod(1)

chown(2), group(4), passwd(4) in *A/UX Programmer's Reference*

NAME

chsh — changes the default login shell

SYNOPSIS

chsh *name* [*shell*]

ARGUMENTS

name

Specifies the name of the user.

shell

Specifies the login shell to be changed to. If no *shell* is specified then the shell reverts to the default login shell /bin/sh. Otherwise only /bin/csh or /bin/ksh can be specified as the shell unless you are the superuser, or the shell named is listed in /etc/shells.

DESCRIPTION

chsh is a command similar to passwd(1) except that it is used to change the login shell field of the password file rather than the password entry.

EXAMPLES

An example of this command is:

```
chsh rusty /bin/csh
```

FILES

/usr/ucb/chsh

Executable file

SEE ALSO

csh(1), passwd(1)

passwd(4), shells(4) in *A/UX Programmer's Reference*

NAME

ci — checks in RCS revisions

SYNOPSIS

ci [-f[*rev*]] [-k[*rev*]] [-l[*rev*]] [-q[*rev*]] [-r[*rev*]] [-u[*rev*]] [-m*msg*] [-n*name*] [-N*name*] [-s*state*] [-t[*txtfile*]] *files*

ARGUMENTS

-f[*rev*]
: Forces a deposit; the new revision is deposited even though it is the same as the preceding one.

files Specifies the RCS files to be checked in.

-k[*rev*]
: Searches the working file for keyword values to determine its revision number, creation date, author, and state (see co(1)), 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 the -r option, except it performs an additional co -l 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.

-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. The ci command prints an error message if *name* is already assigned to another number.

-N*name*
: Works like the -n, option except that it overrides a previous assignment of *name*.

-q[*rev*]
: Specifies quiet mode; diagnostic output is not printed. A revision that is the same as the preceding one is not deposited, unless the -f option is given.

-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 is the owner of the file, and locking is not set to `strict`, then the revision is appended to the trunk.

If a revision number is indicated by *rev*, it must be higher than the latest one on the branch to which *rev* belongs, or *rev* 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.

`-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 standard input, 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 the `-t` option is not given. The prompt is suppressed if standard input is not a terminal.

`-u`[*rev*]
: Works like the `-l` option, except that the deposited revision is not locked. This is useful if one wants to process (that is, compile) the revision immediately after checkin.

DESCRIPTION

`ci` stores new revisions into RCS files. Each filename ending in `,v` is taken to be an RCS file and 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 three ways (see also the example section of co(1)).

(1) Both the RCS file and the working file are given. The RCS filename is of the form *path1*/*workfile*,v and the working filename is of the form *path2*/*workfile*, where *path1* and *path2* are (possibly different or empty) paths and *workfile* is a filename.

(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 for the RCS file, first in the directory ./RCS and then in the current directory.

For ci to work, the caller's login must be on the access list, unless 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(1)). 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 cancels the deposit (if the -q option is given) or asks whether to cancel (if the -q option 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 standard input is not a terminal, ci suppresses the prompt and uses the same log message for all files. Also see the -m option.

The number of the deposited revision can be given by any of the -r, -f, -k, -l, -u, or -q options (see the -r option).

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 the -t option).

An RCS file created by ci inherits the read and execute permissions from the working file. If the RCS file already exists, 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.

STATUS MESSAGES AND VALUES

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 if otherwise.

NOTES

Author: Walter F. Tichy, Purdue University, West Lafayette, IN 47907.

FILES

`/bin/ci`
Executable file

SEE ALSO

`co`(1), `ident`(1), `rcs`(1), `rcsdiff`(1), `rcsintro`(1), `rcsmerge`(1), `rlog`(1)

`sccstorcs`(1M) in *A/UX System Administrator's Reference*

`rcsfile`(4) in *A/UX Programmer's Reference*

Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, September 1982

NAME

clear — clears the terminal screen

SYNOPSIS

clear

DESCRIPTION

clear clears your screen if this is possible. It looks in the environment for the terminal type (TERM) and capabilities string (TERMCAP). If TERMCAP is not found in the environment, it looks in /etc/termcap to figure out how to clear the screen.

EXAMPLES

The command:

```
clear
```

clears the screen.

FILES

/bin/clear
: Executable file

/etc/termcap
: Terminal capabilities file

SEE ALSO

tput(1)

termcap(4), environ(5) in *A/UX Programmer's Reference*

NAME

cmdo — builds command lines interactively

SYNOPSIS

cmdo *command*
cmdo -o *resfile* [-n] [-s] *command*

ARGUMENTS

command
Specifies the command for which you want help with the construction of a command line. If the -o option was given, this argument specifies the name of the command for which you wish to compile a Commando dialog.

-n Specifies that the command not be executed.

-o Compiles the command into the resource file specified by *resfile*. See ''Compiling Resources'' later in the ''Description'' section for additional information.

resfile
Specifies the filename of the compiled resource file.

-s Runs the command silently (the dialog box is not displayed).

DESCRIPTION

cmdo helps you build A/UX command lines using specialized Macintosh dialog boxes. The dialog boxes make it easy to select options, choose files, and access help information, as well as build compound command lines. To build commands with many options and parameters, you may employ several nested dialog boxes.

The contents of dialog boxes are specified in dialog scripts, written in the Commando Script Language. This language is detailed in *A/UX Development Tools*.

The first form of the syntax, the more common of the two, invokes Commando on the specified command. The second form compiles a resource file to speed execution. Commando resource files are fully documented in *MPW Programmer's Workshop*.

Using cmdo

You can run cmdo in two ways. The first is to enter a complete cmdo command line, as follows:

cmdo *command*

The second is to type a partial command line containing only the name of the command within a CommandShell window, then select Commando from the Edit menu. The Command-key equivalent for selecting Commando from the menu is COMMAND-K. This action invokes cmdo on

the last word of the command line.

The first method of invoking `cmdo` executes the command. The second method pastes the command line arguments onto the command line, allowing you to create the desired command line by adding more commands, command options, and arguments. You can then execute the compound command by pressing the RETURN key. Using this method, you can create a command line of many commands piped together, also known as a compound command line.

Search Paths

Commando searches in the following order:

1. Commando searches for resources, then dialog scripts, in the path specified by `$CMDODIR`.
2. Commando searches for resources, then dialog scripts, in the directory within `/mac/lib/cmdo` having the same first letter as the command you are invoking.
3. Commando searches for resources in the path specified by `$PATH`.

Compiling Resources

Commando creates Macintosh dialog boxes from which you select command options and arguments. You can customize the appearance of these dialog boxes by compiling them and modifying them with `cmdo`'s built-in resource editor. For more information, see *MPW Programmer's Workshop*.

EXAMPLES

To display the `ls` Commando dialog box, use this command line:

```
cmdo ls
```

To compile a resource file for an application named `statpack`, use the command line

```
cmdo -o /usr/bin/statpack -r -s statpack
```

The preceding command compiles a resource file and puts it in `/usr/bin`.

LIMITATIONS

When using compiled Commando scripts, make sure the names of the command, dialog script, and resource (prefaced by a `%`, of course) are all be the same.

24-bit login sessions do not support command-line invocation of Commando.

NOTES

CommandShell invokes `cmdo` when you double-click an A/UX command icon for which a dialog script has been written (assuming the script is in the search path). If you execute the command using this method, it executes in a subshell.

24-bit login sessions do support this method of invoking Commando.

FILES

`/mac/bin/cmdo`
The `cmdo` command

`/mac/lib/cmdo/*`
Directories containing source files

SEE ALSO

`CommandShell`(1)

"Commando" in *A/UX Development Tools*

"Creating a Commando Interface for Tools" in *A/UX Programming Languages and Tools, Volume 2*

NAME

cmp — compares two files

SYNOPSIS

cmp [-l] [-s] *file1 file2*

ARGUMENTS

file1

Specifies the file to be compared with *file2*. If this argument is -, the standard input is used.

file2

Specifies the file to be compared with *file1*.

-l Prints the byte number (decimal) and the differing bytes (octal) for each difference.

-s Prints nothing for differing files; return codes only.

DESCRIPTION

The two files are compared. Under default options, cmp makes no comment if the files are the same; if they differ, it announces the byte and line number at which the difference occurred. If one file is a subset of the other, that fact is noted.

EXAMPLES

The command:

```
cmp alpha beta
```

will report if the files are different and at what point they differ, such as:

```
alpha beta differ: char 33, line 2
```

STATUS MESSAGES AND VALUES

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

FILES

/bin/cmp

Executable file

SEE ALSO

bdiff(1), comm(1), diff(1), diff3(1), diffmk(1)

NAME

co — checks out RCS revisions

SYNOPSIS

co [-d*date*] [-j*joinlist*] [-l[*rev*]] [-p[*rev*]] [-q[*rev*]] [-r[*rev*]] [-s*state*] [-w[*login*]] *files*

ARGUMENTS

-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 acceptable formats for *date* are:

```
22-April-1985, 17:20-CDT
2:25 AM, Dec. 29, 1987
Tue-PDT, 1986, 4pm Jul 21
Fri Apr 16 15:52:25 EST 1988
```

The last example illustrates the format produced by ctime(3) and date(1). Most fields in the date and time may be defaulted. co determines the defaults in this order: year, month, day, hour, minute, and second (that is, from 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* specified on the command line must be in quotation marks if it contains spaces.

files Specifies the RCS files to be checked out.

-j*joinlist*

Generates a new revision that is the join of the revisions on *joinlist*. The *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 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* are 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 by the lines <<<<<<< rev1, =======, and >>>>>>> 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 -l option is present, the initial *rev1* is locked.

-l[*rev*]
Locks the checked-out revision for the caller. If omitted, the checked-out revision is not locked. See the -r option 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*]
Specifies quiet mode; error messages are not printed.

-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 a period, (.). 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 that was checked in by the user with login name *login*. If the argument *login* is omitted, the caller's login name is assumed.

DESCRIPTION

co retrieves revisions from RCS files. Each filename 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 in the corresponding working file.

Pairs of RCS files and working files may be specified in three ways (see the EXAMPLES section later in this entry).

(1) Both the RCS file and the working file are given. The RCS filename is of the form *path1/workfile*,v and the working filename is of the form *path2/workfile*, where *path1* and *path2* are (possibly different or

empty) paths and *workfile* is a filename.

(2) Only the RCS file is given. Then the working file is created 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 for the RCS file, first in the directory `./RCS` and then in the current directory.

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 (for example, compiling) need not be locked. A revision checked out for editing and later check-in must normally be locked. Locking a revision currently locked by another user fails. (A lock may be broken with the `rcs`(1) 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, check-in date/time, author, or state. If none of these options is 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. The `co` command always performs keyword substitution, as follows.

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 that 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 updates.

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 as listed. 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 that are 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 are as follows:

$Author$
: 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 filename, 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 filename, the revision number, the author, and the date. Existing log messages are *not* replaced. Instead, the new log message is inserted after $Log:... $. 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.

File Modes

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(1)).

If a file with the name of the working file already exists and has write permission, co cancels the checkout if the -q option is given, or asks whether to cancel if the -q option is not given. If the existing working file is not writable, it is deleted before the checkout.

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
```

STATUS MESSAGES AND VALUES

The RCS filename, the working filename, 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 if otherwise.

LIMITATIONS

The `-d` option 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. In `nroff` and `troff`, this is done by embedding the null-character `\&` into the keyword.

The `-j` option does not work for files that contain lines with a single `..`

NOTES

Author: Walter F. Tichy, Purdue University, West Lafayette, IN 47907.

FILES

`/bin/co`
Executable file

SEE ALSO

`ci`(1), `ident`(1), `rcs`(1), `rcsdiff`(1), `rcsintro`(1), `rcsmerge`(1), `rlog`(1)

`sccstorcs`(1M) in *A/UX System Administrator's Reference*

`rcsfile`(4) in *A/UX Programmer's Reference*

Walter F. Tichy, ''Design, Implementation, and Evaluation of a Revision Control System,'' in *Proceedings of the 6th International Conference on Software Engineering*, IEEE, Tokyo, Sept. 1982

NAME

col — filters text containing printer control sequences for use at a display device

SYNOPSIS

col [-b] [-f] [-p] [-x]

ARGUMENTS

-b Causes col to assume that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

-f Suppresses the movement of text. Although col accepts half-line motions in its input, it normally does not emit them on output. Instead, text that would appear between lines is moved to the next lower full-line boundary. When this option is used, the output from col may contain forward half-linefeeds (ESCAPE-9), but will still never contain either kind of reverse line motion.

-p Causes col to generate unknown escape sequences as regular characters, subject to overprinting from reverse line motions. The use of this option is highly discouraged unless the user is fully aware of the textual position of the escape sequences. Normally, col will ignore any unknown escape sequences found in its input.

-x Does not convert white space to tabs on output wherever possible to shorten printing time.

DESCRIPTION

col reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse linefeeds (ASCII code ESCAPE-7), and by forward and reverse half-linefeeds (ESCAPE-9 and (ESCAPE-8). The col command is particularly useful for filtering multicolumn output made with the .rt command of nroff and output resulting from use of the tbl(1) preprocessor.

The ASCII control characters SO (\016) and SI (\017) are assumed by col to start and end text in an alternate character set. The character set to which each input character belongs is remembered, and on output SI and SO characters are generated as appropriate to ensure that each character is printed in the correct character set.

On input, the only control characters accepted are space, backspace, tab, return, SI , SO , T (\013), and escape followed by 7, 8, or 9. The VT character is an alternate form of full reverse linefeed, included for compatibility with some earlier programs of this type. All other nonprinting characters are ignored.

EXAMPLES

The command:

```
nroff -mm filea | col
```

pipes multicolumn `nroff` output through the `col` filter to enable proper creation of columns.

LIMITATIONS

Cannot back up more than 128 lines.

Allows at most 800 characters, including backspaces, on a line.

Local vertical motions that would result in backing up over the first line of the document are ignored. As a result, the first line must not have any superscripts.

NOTES

The input format accepted by `col` matches the output produced by `nroff` with either the `-T37` or `-Tlp` options. Use `-T37` (and the `-f` option of `col`) if the ultimate disposition of the output of `col` will be a device that can interpret half-line motions, and `-Tlp` otherwise.

FILES

`/usr/bin/col`
Executable file

SEE ALSO

`colcrt`(1), `nroff`(1), `tbl`(1)

NAME

colcrt — filters nroff output for terminal previewing

SYNOPSIS

colcrt [-] [-2] [*file*]

ARGUMENTS

- Suppresses all underlining. It is especially useful for previewing allboxed tables from tbl(1).

-2 Causes all half-lines to be printed, effectively double spacing the output. Normally, a minimal space output format is used that will suppress empty lines. The program never suppresses two consecutive empty lines, however. This option is useful for sending output to the line printer when the output contains superscripts and subscripts that would otherwise be invisible.

file Specifies the file to be passed through the filter.

DESCRIPTION

colcrt provides virtual half-line and reverse line feed sequences for terminals without such capability, and on which overstriking is destructive. Half-line characters and underlining (changed to dashing ''-'') are placed on newlines in between the normal output lines.

EXAMPLES

A typical use of colcrt would be:

```
tbl exum2.n | nroff -mm | colcrt | more
```

LIMITATIONS

Should fold underlines onto blanks even with the - option so that a true underline character would show; if we did this, however, colcrt wouldn't get rid of cu'd underlining completely.

Can't back up more than 102 lines.

General overstriking is lost; as a special case ''|'' overstruck with ''-'' or underline becomes ''+''.

Lines are trimmed to 132 characters.

Some provision should be made for processing superscripts and subscripts in documents that are already double-spaced.

FILES

/usr/ucb/colcrt
Executable file

SEE ALSO

col(1), more(1), nroff(1), troff(1), ul(1)

NAME

colrm — removes columns from a file

SYNOPSIS

colrm *startcol* [*endcol*]

ARGUMENTS

endcol
Specifies the last column to be removed.

startcol
Specifies the first column to be removed.

DESCRIPTION

colrm removes selected columns from a file. Input is taken from standard input. Output is sent to standard output.

If colrm is called with one parameter, the columns of each line are removed starting with the specified column. If colrm is called with two parameters, the columns from the first column to the last column are removed.

Column numbering starts with column 1.

FILES

/usr/ucb/colrm
Executable file

SEE ALSO

awk(1), cut(1), expand(1), sed(1)

NAME

comb — combines SCCS deltas

SYNOPSIS

comb [-c*list*] [-o] [-p*sid*] [-s] *file...*

ARGUMENTS

-c*list*
: Specifies a *list* (see get(1) for the syntax of a *list*) of deltas to be preserved. All other deltas are discarded.

file
: Specifies the SCCS file that will be reconstructed. If a name of - is given, the standard input is read; each line of the input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

-o
: Causes the reconstructed file to be accessed at the release of the delta to be created for each get -e generated; otherwise, the reconstructed file would be accessed at the most recent ancestor (see "SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*). Use of this option may decrease the size of the reconstructed SCCS file. It may also alter the shape of the delta tree of the original file.

-p*sid*
: Specifies the SCCS Identification string (SID) of the oldest delta to be preserved. All older deltas are discarded in the reconstructed file.

-s
: Causes comb to generate a shell script that, when run, will produce a report giving, for each file: the filename, size (in blocks) after combining, original size (also in blocks), and percentage change computed by:

```
100 * (original - combined) / original
```

It is recommended that before any SCCS files are actually combined, one should use this option to determine exactly how much space is saved by the combining process.

DESCRIPTION

comb generates a shell script (see sh(1)) that, when run, will reconstruct the given SCCS files by combining some series of changes. Then the reconstructed files will, hopefully, be smaller than the original files. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, comb behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

The generated shell procedure is written on the standard output.

If no arguments are specified, `comb` will preserve only leaf deltas and the minimal number of ancestors needed to preserve the tree.

EXAMPLES

The command:

```
comb s.file1 > tmp1
```

produces a shell script saved in `tmp1` that will remove from the SCCS-format file, `s.file1`, all deltas previous to the last set of changes, i.e., removes the capability to return to earlier versions.

STATUS MESSAGES AND VALUES

Use `help` for explanations.

LIMITATIONS

The `comb` command may rearrange the shape of the tree of deltas. It may not save any space; in fact, it is possible for the reconstructed file actually to be larger than the original.

FILES

`/usr/bin/comb`
: Executable file

`s.COMB`
: Reconstructed SCCS file

`comb`*???*
: Temporary file

SEE ALSO

`admin`(1), `delta`(1), `get`(1), `help`(1), `prs`(1), `sh`(1)

`sccsfile`(4) in *A/UX Programmer's Reference*

"SCCS Reference" in *A/UX Programming Languages and Tools, Volume 2*

NAME

comm — selects or rejects lines common to two sorted files

SYNOPSIS

comm [- [1] [2] [3]] *file1 file2*

ARGUMENTS

- Specifies the standard input.

1 Suppresses printing of the first column.

2 Suppresses printing of the second column.

3 Suppresses printing of the third column.

file1
Specifies the first sorted file.

file2
Specifies the second sorted file.

DESCRIPTION

comm reads *file1* and *file2* which should be ordered in ASCII collating sequence (see sort(1)), and produces a three-column output: lines only in *file1*, lines only in *file2*, and lines in both files.

EXAMPLES

The command:

```
comm -12 filea fileb
```

prints only the lines common to filea and fileb. The command:

```
comm -23 filea fileb
```

prints only lines in the first file but not in the second. The command:

```
comm -123 filea fileb
```

is not an option, as it suppresses all output. The command:

```
comm -3 filea fileb
```

prints only the lines that differ in the two files.

FILES

/usr/bin/comm
Executable file

SEE ALSO

bdiff(1), cmp(1), diff(1), diff3(1), diffmk(1), sort(1), uniq(1)

NAME

CommandShell — manages command-interpretation windows and moderates access to the A/UX console window

SYNOPSIS

CommandShell [-b *macsysinit-pid*] [-q] [-u]

ARGUMENTS

-b *macsysinit-pid*
Starts CommandShell in a background layer, without any windows, at system startup. With this option, CommandShell displays the A/UX console window but does not accept any other window-management commands. To display the console window, choose CommandShell from the Application menu (or Apple menu if it is not already open) and then choose A/UX System Console from the Window menu. No command-interpretation windows can be displayed. For further information on kernel errors or requests for input, see ''The A/UX Console Window'' in the ''Description'' section. Replace *macsysinit-pid* with the process ID of macsysinit (see brc(1M)) as an argument. After taking control of the system console, CommandShell sends this process a message telling it to exit. This message signals the continuation of all the remaining startup processes. Alert boxes are displayed as necessary to notify users of the need for input.

-q Supresses the Quit menu item in the File menu; brings CommandShell into the foreground.

-u Specifies that a user has logged in and started CommandShell in a background layer. When CommandShell is made the active application, the user's preferred (or the default) CommandShell window layout is established. The default task is to open a single CommandShell window. When you start CommandShell this way, you can choose Save Preferences from the File menu to save your preferences.

DESCRIPTION

CommandShell provides a Macintosh user interface to A/UX users. The available CommandShell menus are described in ''Menu Items'' later in the ''Description'' section. Within CommandShell windows, you can enter A/UX command lines for processing by one of the available shells.

You can use the Macintosh copy and paste operations to enter A/UX commands in a CommandShell window. You can copy text from any previously entered command lines in the same window as well as any text available in other windows (including other Macintosh application windows). Because the CommandShell windows are scrollable, you can

make previous commands or their output available for copying.

You can also build an A/UX command line semiautomatically by using the Commando dialog boxes. Enter the command on the CommandShell command line; then choose Commando from CommandShell's Edit menu or press COMMAND-K. You can also use the `cmdo` command (see `cmdo`(1)).

CommandShell Windows

When you start CommandShell, one window is displayed by default unless you have saved a previous window layout that specifies more than one window. The default window is titled "CommandShell 1." You can create more windows by choosing New from the File menu or by pressing COMMAND-N. Each time you perform one of these actions, a new window appears in front of the existing window or windows. The title bar of each new window is numbered in sequence according to the order of its creation. Normally you can create up to 15 windows.

When you create a new window, it appears in front of, and slightly to the right of and below, the last current window. You can use the titling commands to view the contents of all the windows. For information on specific titling commands, see "Window Menu" later in the "Description" section.

The A/UX Console Window

Besides managing command-interpretation windows, CommandShell moderates access to the A/UX console window. This window is one of the places where the A/UX environment and the Macintosh desktop environment meet. Kernel error messages are routed to this window so as not to disturb the bitmapped display of Macintosh applications.

The Macintosh user interface is an integral part of the A/UX boot process, supported in part by the A/UX console window of CommandShell. This special window is the place where all boot messages appear and where you enter responses in the event that one of the boot processes requires your input.

System-startup messages. During the boot process, CommandShell disables many of the functions in its menus. The status messages that are normally directed to the system console during startup (many of which are useful only to the system administrator) are directed to the A/UX console window. To inspect the boot messages generated during the last system startup, choose A/UX System Console from the Window menu of CommandShell.

To view the A/UX console window, perform these steps:

1. If CommandShell is not active, choose it from the Application menu (or Apple menu if it is not open).
2. Choose A/UX System Console from the Window menu.
3. Use the scroll bar as needed to view the contents of the window.

Notification of messages. Processes that run as part of a startup script, such as `/etc/sysinitrc`, or as another part of the booting process may occasionally require user input. For example, suppose you add an Ethernet card to your system. Then suppose that, while rebooting, the system needs to request address information about the new card. A/UX does not display a prompt asking for this information. Instead, an alert box, telling you that an A/UX process requires input, appears in front of the A/UX boot progress bar. Click OK in the alert box to cause the A/UX console window to appear and the alert box to disappear. The A/UX console window contains messages prompting you for input. You can now enter information in the window. At all other times, the window is for reading purposes only.

This alerting process is called a ''notification system.'' A similar notification system has been created for the handling of A/UX kernel messages. The system displays alert boxes that encourage you to inspect the text of the A/UX error messages in the normally hidden A/UX console window.

Here is what you should do in response to this form of notification:

1. After reading the alert box, click OK.
2. Make CommandShell the selected application if it is not already in control of the active window.
3. Choose A/UX System Console from the Window menu of CommandShell. The normally hidden A/UX console window is displayed; it contains all of the error messages generated since the system was last booted. If you scroll upward in the window, you can see the old messages. CommandShell also lets you respond to any prompts for input.

If the error listed in the A/UX console window concerns system or network performance (such as a number of retries before successful transmission of a network packet), then no further corrective action is required. Sometimes, however, the error message may indicate a serious error condition, such as one of the following:

```
file system full
file system corrupt
```

```
fork failed: too many processes
```

In these cases, you may lose data if the error condition persists.

Changing Notification Preferences. You can change the way the system notifies you that the A/UX console window has a message for you when CommandShell is not the active application. To set notification preferences, make CommandShell active and choose Notification Levels from the Preferences menu. In response, the system displays a dialog box in which you can select to be notified by an alert box, a blinking icon in the menu bar, or both. User preferences are normally stored in the `.cmdshellprefs` files located in your home directory. (See "Managing CommandShell Preferences" later in the "Description" section.) The default notification is an alert box because console messages may indicate a fundamental system problem that you should know about immediately.

Leaving CommandShell

It is not advisable to quit CommandShell because you will lose the ability to inspect the console window and enter responses to any prompts for information sent there.

The next best thing to a Quit function is the option to choose Hide CommandShell Windows in the Applications menu.

Terminal Attributes

The hardware used as the means for accessing a UNIX system is often a keyboard and a character-oriented display. Such display devices are often called ttys or terminals. When you have a CommandShell window open, your terminal is used to emulate a conventional terminal device, including various display features and modes.

For example, most terminal devices will respond to certain sequences of text as commands requesting the use of a special display mode, such as the display of underlined text. However, the command sequence required varies slightly from one terminal (or manufacturer) to another.

For the default type of terminal that CommandShell emulates, VT102, additional command sequences exist to allow the selection of cursor type, tab stops, cursor position, line height, number of display lines, number of display columns, and so on. See `vt102`(7) for a summary of this information. (You could also look at one of the commercial books that more fully describe the escape and control sequences for VT102-style terminals.)

Furthermore, the particular VT102 emulation that CommandShell provides offers support beyond the standard control and escape sequences to permit you to control CommandShell windows, such as resizing or retitling them based upon a control sequence. For these special features that exceed

VT102 specifications, see ''Window Control Sequences'' later in the ''Description'' section.

Note that only an A/UX process (usually a shell) running in the active window (the window with horizontal lines in the title bar) receives key sequences generated through the keyboard inputs. Other CommandShell windows must first be activated before keyboard input can be read by them.

To allow you to run a UNIX program that expects to send command sequences for a particular type of terminal device, CommandShell can be set to emulate such third-party devices in open CommandShell windows. To help identify the terminal type, conventional UNIX systems require you to place special values inside the `TERM` shell variable (see `termcap`(4) and `terminfo`(4)). With A/UX, you need to inform CommandShell and the shell of the desired setting by setting the `TERM` variable.

The interface supported by character-display terminals may seem, at first impression, to be inferior to a graphic display simply because it is old-fashioned.

However, consider the ways in which you can perform file-manipulation operations such as ''delete file.'' The desktop metaphor helps depict the action of file deletion graphically because you drag the file icon to the Trash icon. Using the old-fashioned approach, you open a CommandShell window to run the command `rm`. The old-fashioned way is harder to learn and less intuitive in the simple case. However, the graphic interface can be tedious and laborious when you need to perform a more sophisticated deletion. Consider the task of removing from the current directory only those files named a particular way. For example, say that you want to delete files only if their names end with ''.tmp.'' In this case, the easier way to delete them is to use a command line, such as this one:

```
rm *.tmp
```

Menu Items

CommandShell displays menus titled File, Edit, Window, Fonts, Commands, Keys, and Preferences in the menu bar at the top of the screen, plus the Apple menu at the left end of the menu bar. To choose a menu item, position the pointer on the menu title, press and hold down the mouse button, and move the pointer to the menu. Release the mouse button when the pointer highlights the desired item.

You can choose many menu items from the keyboard by holding down the COMMAND key (not the CONTROL key) and typing a character. This COMMAND-key equivalent is shown beside the menu item. You can enter the COMMAND-key equivalents as lowercase letters; you don't need to hold down SHIFT as well.

The sections that follow describe the actions performed by the CommandShell menu items.

Apple Menu. At the left end of the menu bar, the Apple symbol is the title of a menu that contains some general Macintosh desk accessories and some menu items specifically related to CommandShell. These menu items are related to CommandShell:

About CommandShell
: Displays a dialog box that gives version information.

CommandShell
: Makes CommandShell the active application and makes the CommandShell window that was most recently active the active window once again.

File Menu. The menu items in the File menu allow you to create and close windows, to select printing options, and so on. The File menu contains the following items:

New
: Creates a new window. The windows are numbered sequentially according to the order of their creation. The COMMAND-key equivalent for the New menu item is COMMAND-N.

Open
: Launches a UNIX command or launches an editor if the highlighted file is a text file. The COMMAND-key equivalent for the Open menu item is COMMAND-O.

Close
: Closes the active window. Before you close a window, make sure that you write the contents of the window to a disk if you want to save your work. The COMMAND-key equivalent for the Close menu item is COMMAND-W.

Save Selection
: Saves the contents of a CommandShell window in an A/UX file. The text you want to save must be selected (highlighted).

Page Setup
: Displays a dialog box that lets you set the paper size, orientation, and reduction or enlargement for subsequent printing opertions.

Print Selection
: Prints selected text from the active window. Use the Chooser desk accessory, available in the Apple menu, to specify which printer to use. Use the Page Setup menu item just described to specify paper size, orientation, and scale.

Close All Windows
: Closes all open windows at once. Before you close the windows, make sure that you write the contents of each window to a disk if you want to save your work. If you don't write the contents to a disk, they are lost.

Quit
: Quits CommandShell, closing any windows that are open.

Edit Menu. The menu items in the Edit menu let you move text around and perform certain global formatting actions. The Edit menu contains these items:

Undo
: Reverses the most recent text change. If you choose Undo a second time, the change is reinstated. The COMMAND-key equivalent for the Undo menu item is COMMAND-Z.

Cut
: Copies the currently selected text in the active window to the Clipboard and then deletes it from the window. This menu item is used with desk accessories only; unless a desk accessory is active, it is disabled. The COMMAND-key equivalent for the Cut menu item is COMMAND-X.

Copy
: Copies the currently selected text in the active window to the Clipboard without deleting it from the window. The COMMAND-key equivalent for the Copy menu item is COMMAND-C.

Paste
: Inserts the contents of the Clipboard at the current cursor location. The COMMAND-key equivalent for the Paste menu item is COMMAND-V.

Clear
: Deletes the currently selected text from the active window. This menu item is used with desk accessories only; unless a desk accessory is active, it is disabled. The keyboard equivalent for the Clear menu item is DELETE.

Select All
: Selects the entire document shown in the active window. The COMMAND-key equivalent for the Select All menu item is COMMAND-A.

Commando
: Builds commands semiautomatically. Choose this menu item after entering the command name at the beginning of a line. A dialog box appears, depicting all the features of the command so

that you can select the ones you want to use. When you close the dialog box, the command line that you started to specify before choosing Commando is changed to include all of the command options and arguments that you generated with the help of the Commando dialog box. The COMMAND-key equivalent for the Commando menu item is COMMAND-K.

Window Menu. The menu items in the Window menu help you arrange and display CommandShell windows. The menu is divided into three parts. The upper part of the menu contains menu items that help you arrange windows in various formats. The middle part contains menu items that help you size and order the windows. The lower part contains a list of all windows currently available in CommandShell. When you choose one of the window names in the lower part of the menu, CommandShell makes the corresponding window active. The names of currently available windows are listed in the order in which they were opened.

The menu items in the top part of the menu do the following tasks:

Tile Positions windows in a right-to-left, then top-to-bottom sequence. You must have more than one window open on the desktop to use this menu option. The COMMAND-key equivalent for the Tile menu item is COMMAND-T.

Tile Horizontal
Positions windows from top to bottom on the screen in the order in which they were opened. The windows are enlarged to the width of the screen. The height of each window is adjusted to accommodate the number of windows.

Tile Vertical
Positions windows from left to right on the screen in the order in which they were opened. The windows are enlarged to the height of the screen. The width of each window is adjusted to accommodate the number of windows.

Standard Positions
Repositions the windows in the original stacked order, from front to back.

The items in the middle part of the menu do the following tasks:

Standard Size
Resizes a window to its original dimensions. The COMMAND-key equivalent for the Standard Size menu item is COMMAND-S.

Full Height
Enlarges a window to the full height of the screen. The COMMAND-key equivalent for the Full Height menu item is COMMAND-F.

Zoom Window
: Enlarges the window to the full height and width of the screen. You can return a window to its previous size by choosing the Zoom Window menu item again. The COMMAND-key equivalent for the Zoom Window menu item is COMMAND-backslash.

Hide *window-name*
: Makes the specified window (*window-name*) temporarily disappear. The window is no longer visible, but it is still available. To display a window that has been hidden, choose this command from the Window menu again. The window reappears, as the active window. The COMMAND-key equivalent for the Hide *window-name* menu item is COMMAND-H.

Show All Windows
: Displays all windows that have been hidden.

Last Window
: Makes the previously active window the active window once again, making it visible if it was hidden. Choosing the command again returns the windows to their original states. The COMMAND-key equivalent for the Last Window menu item is COMMAND-L.

Rotate Window
: Matches the window that is at the back of the window stack to the active window. The COMMAND-key equivalent for the Rotate Window menu item is COMMAND-R.

The menu items in the lower part of this menu are the names of all currently available CommandShell windows. When you choose a name, the corresponding window becomes active and moves to the front. The A/UX console window is always included in this list.

A/UX System Console
: Makes the A/UX console window the active window. This window displays console messages. The COMMAND-key equivalent for the A/UX System Console menu item is COMMAND-0 (zero).

Fonts Menu. The Fonts menu lets you choose the font and the point size of text entered or displayed in the active CommandShell window.

The last item in the Fonts menu, Other, displays a dialog box that allows you to enter the exact point size you desire.

Commands Menu. You use the menu items in the Commands menu to choose default settings for recording information and to clean up the screen.

Don't Record Lines Off Top/Record Lines Off Top
: Toggles between recording a preset number of lines that have scrolled

off of the screen or deleting these lines. When you start CommandShell, it is set to record a preset number of lines as they scroll out of view. If you do not want to store the lines for possible review later, you can stop the recording of lines for the active CommandShell window by choosing Don't Record Lines Off Top. To begin recording lines again, choose Record Lines Off Top. The lines are again recorded as they scroll out of view at the top of the window.

Clear Lines Off Top
: Erases recorded lines and makes them no longer available for review within the active window. The scroll bar disappears from the active CommandShell window.

Redraw Screen
: Cleans up the screen if text output affects the bitmapped display.

Preferences Menu. You use the menu items in the Preferences menu to specify how you want to be notified of system messages, to choose your default window settings, and to set your preferred window configuration. For more complete descriptions of the dialog boxes that are associated with these menu items, refer to the CommandShell chapter in *A/UX Essentials* about CommandShell.

Notification Levels
: Sets the notification level for console messages. A dialog box appears in which you specify how you want to be notified of console messages. The choices are an alert box, a blinking icon in the menu bar, or both.

New Window Settings...
: Specifies the default title prefix, point of origin of the window cascade, window size, font name, font size, and number of lines saved off the top of the window. A dialog box appears in which you set these specifications. The Set Emulation button leads to a set of control-panel dialog boxes for setting terminal emulation parameters.

Active Window Settings...
: Specifies the settings for the active window. A dialog box appears allowing you to specify the title, size, and position of the window; a setting that determines whether to record lines that scroll out of view; and an initial command to run in the window when it opens. The Set Emulation button leads to a set of control-panel dialog boxes for setting terminal emulation parameters.

Save Preferences
: Saves all window settings, layout information, and notification-level settings.

Restore From Preferences
: Restores window settings and layout to the ones specified in the preferences file. This command also activates any saved windows that have been closed and runs an initial command in windows that do not already have a command running.

Keys Menu. This menu contains two items that display submenus that are not formed of the usual list of commands, but rather are graphic palettes that represent key buttons.

Key pad
: Displays a keypad of numeric keys, in which clicking a "key" is equivalent to pressing the corresponding number key on the numeric keypad of a keyboard. The corresponding value is entered on the command line at the current cursor position.

Cursor Keys
: Displays a keypad of the arrow keys, in which clicking a "key" is the equivalent of pressing the corresponding arrow key on a keyboard. Each selection from this palette alters the position of the cursor in the currently active CommandShell window.

Managing CommandShell Preferences

Preferences are normally saved in the `.cmdshellprefs` file located in the home directory. To maintain more than one set of preferences, you can establish a different filename for the file in which preferences are stored. For example, to save one set of preferences (window sizes and so forth) for use with a large display device and another set of preferences for use with a smaller display device, you can reset the CommandShell variable that controls the file in which these settings are maintained. The name of this variable is `CMDSHELLPREFS`. You can set this variable to something other than `.cmdshellprefs`. When you reset the variable, you should assign it a filename relative to your home directory.

Window Control Sequences

When the emulation mode has been set to VT102, you can provide additional control sequences besides those that are standard features of VT102 devices.

These additional sequences allow you to control CommandShell windows through a means other than its interactive menus and dialog boxes. Here is a list of these control sequence formats. `\E` is used to represent the ASCII code (27) for the ESC key.

`\E[1t`
: Makes the window associated with the process that sends this sequence the active window (displaying it if it was hidden).

\E[4;*height*;*width*t
Resizes the window associated with the process that sends this sequence in terms of the pixel amounts given in place of *height* and *width*.

\E[5t
Moves the window associated with the process that sends this sequence to the front, without affecting the status of CommandShell as the selected application.

\E[6t
Moves the window associated with the process that sends this sequence to the back of any other CommandShell windows, without affecting the status of CommandShell as the selected application.

\E[6t
Refreshes the window associated with the process that sends this sequence, without affecting the status of CommandShell as the selected application.

\E]l*title*\E\
\E]L*title*\E\
Retitles the window associated with the process that sends this sequence, causing it to be named *title*.

Some control sequences you can send are not used to directly control a CommandShell window, but instead request information about the window. Here is a list of these sequence formats for querying windows. (\E is used to represent the ASCII code (27) for the ESC key.

\E11t
Causes CommandShell to generate a text string on the standard input of the process that sent the string. The generated string indicates whether the window originally associated with a process is still open. The sequences that CommandShell can generate in response are as follows:

\E[1t
The CommandShell window is still open.

\E[2t
The CommandShell window is closed.

\E13t
Causes CommandShell to generate a text string on the standard input of the process that sent the string. The generated string indicates the current location of the window originally associated with the process, in the following format:

\E[3;*topmost-pixel*;*leftmost-pixel*t

`\E14t`
Causes CommandShell to generate a text string on the standard input of the process that sent the string. The generated string indicates the current size of the window originally associated with the process, in the following format:

`\E[4;`*height-in-pixels*`;`*width-in-pixels*`t`

`\E18t`
Causes CommandShell to generate a text string on the standard input of the process that sent the string. The generated string indicates the current size of the window originally associated with the process, in the following format:

`\E[8;`*character-rows*`;`*character-columns*`t`

`\E21t`
Causes CommandShell to generate a text string on the standard input of the process that sent the string. The generated string indicates the current title of the window originally associated with the process, in the following format:

`\E[l`*title*`\`

FILES

`/mac/bin/CommandShell`
Executable file

`/mac/bin/%CommandShell`
Resource fork file

`$HOME/.cmdshellprefs`
Default preferences file

`$HOME/System Folder/Extensions/VT102`
Link file

`/mac/lib/SystemFiles/shared/Extensions/VT102`
Communications Toolbox binary file

`/mac/sys/System Folder/Extensions/VT102`
Link file

SEE ALSO

`cmdo`(1)

`termcap`(4) in *A/UX Programmer's Reference*

`brc`(1M), `startmac`(1M), `startmsg`(1M), `StartMonitor`(1M) in *A/UX System Administrator's Reference*

`vt102`(7) in *A/UX Programmer's Reference*

A/UX Essentials

NAME

compact, uncompact, ccat — compress and uncompress files

SYNOPSIS

compact [*name*]...

uncompact [*name*]...

ccat [*file*]...

ARGUMENTS

file Specifies the file to be displayed. If this argument is not given, the standard input is compacted or uncompacted to the standard output.

name
Specifies the files to be compacted or uncompacted.

DESCRIPTION

compact compresses the named files using an adaptive Huffman code. If no filenames are given, the standard input is compacted to the standard output. compact operates as an on-line algorithm. Each time a byte is read, it is encoded immediately according to the current prefix code. This code is an optimal Huffman code for the set of frequencies seen so far. It is unnecessary to prefix a decoding tree to the compressed file since the encoder and the decoder start in the same state and stay synchronized. Furthermore, compact and uncompact can operate as filters. In particular, the command sequence

```
| compact | uncompact |
```

operates as a (very slow) no-op.

When an argument *name* is given, it is compacted and the resulting file is placed in *file*.C. *file* is unlinked. The first two bytes of the compacted file code the fact that the file is compacted. This code is used to prohibit recompaction.

The amount of compression to be expected depends on the type of file being compressed. Typical values of compression are: Text (38%), Pascal Source (43%), C Source (36%) and Binary (19%). These values are the percentages of file bytes reduced.

uncompact restores the original file from a file compressed by compact.

ccat cats the original file from a file compressed by compact, without uncompressing the file.

LIMITATIONS

The last segment of the filename must contain fewer than thirteen characters to allow space for the appended `.C`.

FILES

`/usr/ucb/compact`
: Executable file

`/usr/ucb/uncompact`
: Executable file

`/usr/ucb/ccat`
: Executable file

`*.C`
: Compacted files

SEE ALSO

`pack`(1)

Gallager, Robert G., *Variations on a Theme of Huffman*, I.E.E.E. Transactions on Information Theory, vol. IT-24, no. 6, November 1978, pp. 668-674

NAME

compress, compressdir, uncompress, uncompressdir, zcat, zcmp, zdiff, zmore — compress files and directories as well as expand them; support concatenation, browsing, and file-comparing operations upon compressed files

SYNOPSIS

compress [-b *maxbits*] [-c] [-f] [-v] [-V] [*file*]...

compressdir [*compress-flag*]... [*directory*]...

uncompressdir [*uncompress-flag*]... [*directory*]...

uncompress [-c] [-f] [-v] [-V] [*file*]...

zcat [-V] [*file*]...

zcmp [*cmp-option*]... *file1* [*file2*]

zdiff [*diff-option*]... *file1* [*file2*]

zmore [*file*]...

ARGUMENTS

-b *maxbits*
: Uses *maxbits* as the maximum number of bits to use in codes when compressing file. The compress command uses a modified Lempel-Ziv algorithm according to which common substrings in the file are first replaced by 9-bit codes 257 and up. When code 512 is reached, the algorithm switches to 10-bit codes and continues to use more bits until the limit, specified by the -b option, is reached (default 16). The *maxbits* specification 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 attained, compress periodically checks the compression ratio. If it is increasing, compress continues to use the existing code dictionary. However, if the compression ratio decreases, compress discards the table of substrings and rebuilds it from scratch. This allows the algorithm to adapt to the next ''block'' of the file.

-c
: Makes compress or uncompress write to the standard output; no files are changed. The nondestructive behavior of zcat is identical to that of uncompress -c.

cmp-option
: Specifies one of the command options for the cmp command as described in cmp(1).

compress-flag
: Specifies one of the command options available for compress.

diff-option
Specifies one of the command options for the `diff` command as described in `diff`(1).

directory
Specifies the starting directory within which all files are compressed or uncompressed, including those in nested directories.

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

file Specifies the file to be affected by a compression or uncompression operation.

file1
Specifies one of the pair of files to be compared by `zcmp` or `zdiff`.

file2
Specifies one of the pair of files to be compared by `zcmp` or `zdiff`.

uncompress-flag
Specifies one of the command options available for `uncompress`.

`-v` Prints a message yielding the percentage of reduction for each file compressed.

`-V` Prints the current version and compile options on the standard output.

DESCRIPTION

`compress` and `compressdir` reduce the size of the named files, or the files residing under the named directories, using adaptive Lempel-Ziv coding. Whenever possible, each file is replaced by one with the extension `.Z`, while keeping the same ownership modes, access, and modification times. If no files are specified along with `compress`, the standard input is compressed to the standard output. If no directories are specified along with `compressdir`, the compression is applied to all files starting with the current directory.

Compressed files can be restored to their original form using `uncompress` and `uncompressdir` along with appropriate arguments. Compressed files can be converted to an uncompressed data stream for viewing onscreen (or output redirection) by `zcat` (see `cat`(1)) and `zmore` (described in greater detail later in a subsequent subsection). Compressed files can be compared with one another and reports of differences can be produced by `zcmp` and `zdiff` (described later).

Note that the -b option is omitted for uncompress, since the *maxbits* parameter specified during compression is encoded within the output, along with a magic number to ensure that neither decompression of random data nor recompression of compressed data is attempted.

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

Exit status is normally 0; if the last file is larger after (attempted) compression, the status is 2; if an error occurs, exit status is 1.

zmore Filter

zmore is a filter which allows examination of compressed text files one group of lines at a time. It pauses after the first screenful of text and displays --More-- at the bottom of the screen. With each press of the RETURN key, one more line is displayed. With each press of the SPACE BAR, another screenful of text is displayed.

The zmore program looks in the file /etc/termcap to determine terminal characteristics, and to determine the default window size. For a terminal capable of displaying 24 lines, the default window size is 22 lines for zmore.

Besides entering spaces and return characters to control operation, you can enter other keys and key combinations. You can consider the --More-- message to be the command prompt for zmore. If for some reason you do not see this message (possibly after command interruption), you can type the erase character (normally by pressing DELETE) to cause the prompt to be redisplayed.

It can be somewhat difficult to cancel a zmore command sequence under construction. First, the zmore command doesn't display the command entry under construction. (The exceptions are the string seek commands and the escape-to-shell command. These commands cause your command entry to be displayed.) Second, zmore automatically recognizes when a command has been fully formed and immediately runs it (without your pressing RETURN at its end). To cancel a partially entered command sequence, you can blindly type the line kill character (normally by pressing CONTROL-U).

The following list describes the effect of each of the zmore commands sequences. If an optional argument (enclosed in brackets) is not given, it normally defaults to 1.

[*numlines*]SPACE BAR
Displays the specified number of lines from the input file that occur after the last one that had been displayed previously (or another screenful if the optional argument is omitted).

[*numlines*]CONTROL-D
[*numlines*]d
Display 11 more lines (a ''scroll''). If *numlines* is given, then the scroll size is the specified amount.

[*numlines*]z
Displays the specified number of lines from the input file that occur after the last one that had been previously displayed (same as a space command), except that if *numlines* is present, it becomes the new window size. Note that the window size reverts back to the default at the end of the current file.

[*numlines*]s
Skips the specified number of lines, then displays the next screenful of lines

[*numscreens*]f
Skips the specified number of text screens, then displays a screenful of lines

q
:q
Q
:Q
Cause zmore to quit displaying the current file, skipping to the next file (if any).

e Causes zmore to quit rather than continue to the next input file whenever the prompt

--More--(Next file: *file*)

appears.

= Displays the current line number.

[*nth-occurrence*]/*regular-expr*
Searches for the nth occurrence of the regular expression specified. If the pattern is not found, *zmore* goes on to the next file (if any). Otherwise, text is displayed starting two lines before the place where the target string was found. The erase and kill characters may be used to edit the regular expression. Erasing backwards past the first column cancels the search command.

[*nth-occurrence*]`/n`
Searches for the nth occurrence of the previously entered regular expression (see preceding item).

`!`*command*
Invokes a shell to run the specified command. Any exclamation mark character (`!`) inside the specified command is replaced with the previous shell command specified. To allow a literally interpreted exclamation mark to be entered as part of the command line, the sequence `\!` is replaced by `!`.

. (period character) Repeats the previous command.

To interrupt a display action at any time, you can type the quit character (normally by pressing CONTROL-\). This causes `zmore` to stop sending output, but it still displays the usual prompt afterwards prompting you for one of the commands listed preceding. Note that some text is often lost when a display operation is interrupted.

The terminal is set to `noecho` mode by this program so that the output can be continuous. As a result, the characters you type do not normally appear on your display screen (so you type commands blindly).

If the standard output for `zmore` is redirected to a place other than the terminal, then `zmore` functions like `zcat`, except that a header is printed before each file.

`zcmp` and `zdiff` File Comparers

`zcmp` and `zdiff` are used to report about the differences between two files without requiring you to uncompress them first. These commands operate the same way that the corresponding commands `cmp` and `diff` operate. The command options are passed along to the `cmp` and `diff` programs, which are appropriately called by `zcmp` and `zdiff`.

If only one file is specified, the files that it attempts to compare are the named one (*file1*) and another by the same name but with `.Z` added.

If two files are specified, then they are temporarily uncompressed (as necessary) and submitted to the appropriate command (`cmp` or `diff`). The exit status is the exit status from `cmp` or `diff`.

STATUS MESSAGES AND VALUES

Messages from the `cmp` or `diff` programs refer to temporary filenames instead of those specified.

The following error messages may appear for the `compress`, `compressdir`, `uncompress`, or `uncompressdir` commands:

`Usage: compress [-dfvcV] [-b maxbits] [file...]`
Invalid options were specified on the command line.

`Missing maxbits`
maxbits must follow `-b`.

file`: not in compressed format`
The file specified to `uncompress` has not been compressed.

file`: already has .Z suffix -- no change`
The file is assumed to be already compressed. Rename the file and try again.

file`: filename too long to tack on .Z`
The file cannot be compressed because its name is longer than 12 characters. Rename and try again.

file `already exists; do you wish to overwrite (y or n)?`
Respond `y` if you want the output file to be replaced; `n` if not.

The following general error messages may appear regarding the compression or uncompression process:

file`: compressed with` *xx* `bits, can only handle` *yy* `bits`
file was compressed by a program that could deal with more bits than the compress code on this machine. Recompress the file with smaller *maxbits*.

`uncompress: corrupt input`
A `SIGSEGV` violation was detected, which usually means that the input file has been corrupted.

`Compression:` *xx.xx*`%`
Percentage of the input saved by compression. (Relevant only for `-v`.)

`-- not a regular file: unchanged`
When the input file is not a regular file (for example, a directory), it is left unaltered.

`--`*has* `xx` *other links: unchanged*
The input file has links; it is left unchanged. See `ln`(1) for more information.

`-- file unchanged`
No savings is achieved by compression. The input remains virgin.

LIMITATIONS

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

FILES

`/etc/termcap`
Terminal data base file

`/usr/ucb/compress`
Executable file

`/usr/ucb/uncompress`
Executable file

`/usr/ucb/zcat`
Executable file

SEE ALSO

`cat`(1), `cmp`(1), `compact`(1), `diff`(1), `more`(1), `pack`(1)

Terry A. Welch, ''A Technique for High Performance Data Compression,'' *IEEE Computer*, Vol. 17, No. 6 (June 1984), pages 8-19

See compress(1)

NAME

conv — swaps bytes in COFF files

SYNOPSIS

conv [-] [-a] [-o] [-p] [-s] -t*target* file...

ARGUMENTS

- Reads *file*s from standard input.

-a Produces the output file in the old archive format, if the input file is an archive.

file Specifies the converted file.

-o Produces the output file in the UNIX 6.0 (Version 6) portable archive format, if the input file is an archive.

-p Specifies the UNIX V.0 random access archive format. This is the default.

-s Byte-swaps all bytes in object file. This is useful only for 3B20 object files which are to be swab-dumped from a DEC machine to a 3B20.

-t*target*
Converts the object file to the byte ordering of the machine (target) to which the object file is being shipped. This may be another host or a target machine. Legal values for *target* are: pdp, vax, ibm, i86, x86, b16, n3b, m32, and m68k.

DESCRIPTION

conv converts object files from their current format to the format of the *target* machine. The converted *file* is written to *file*.v.

The conv command can be used to convert all object files in common object file format. It can be used on either the source (sending) or target (receiving) machine.

The conv command is meant to ease the problems created by a multihost cross-compilation development environment. The conv command is best used within a procedure for shipping object files from one machine to another.

The conv command will recognize and produce archive files in three formats: the UNIX pre-V.0 format, the V.0 random access format, and the 6.0 portable ASCII.

EXAMPLES

The following shows how this command is used:

```
echo *.out | conv - -t m68k
```

STATUS MESSAGES AND VALUES

All messages for the `conv` command are intended to be self-explanatory. Fatal messages on the command lines cause termination. Fatal messages on an input file cause the program to continue to the next input file.

WARNINGS

The `conv` command does not convert archives from one format to another if both the source and target machines have the same byte ordering.

FILES

`/bin/conv`
: Executable file

NAME

cp — copies files

SYNOPSIS

cp [-i] [-r] *file1 file2*

cp [-i] [-r] *file... directory*

ARGUMENTS

directory
Specifies the directory into which *file* will be placed.

file Specifies the file that will be copied into *directory*.

file1
Specifies the file to be copied onto *file2*.

file2
Specifies the file into which *file1* will be copied.

-i Prompts the user with the name of the file whenever the copy will cause an old file to be overwritten. An answer of y will cause cp to continue. Any other answer will prevent it from overwriting the file.

-r Copies each subtree rooted at that name (if any of the source files are directories). The destination must be a directory.

DESCRIPTION

cp copies *file1* onto *file2*. The mode and owner of *file2* are preserved if it already existed; otherwise, the mode of the source file is used (all bits set in the current umask value are cleared).

In the second form, one or more *file*s are copied into the *directory* with their original filenames.

The cp command refuses to copy a file onto itself.

WARNINGS

The cp command does not copy the description of special files, but attempts to copy the contents of the special files. This often occurs when using the -r option for a recursive copy. For example, cp will hang when trying to copy a named pipe or tty device. When a disk node is being copied, the contents of the disk partition will be copied. To copy the description of the special files, use cpio(1).

FILES

/bin/cp
Executable file

SEE ALSO

cat(1), pr(1), mv(1), rcp(1C)

NAME

cpio — copies files to or from a cpio archive

SYNOPSIS

cpio -o [a] [c] [B] [F] [v]

cpio -i [6] [b] [B] [c] [d] [f] [m] [r] [s] [S] [t] [u] [v] [*patterns*]

cpio -p [a] [d] [l] [m] [u] [v] *directory*

ARGUMENTS

-6 Processes an old (that is, UNIX System Sixth Edition format) file. Useful only with -i option.

-a Resets access times of input files after they have been copied.

-b Swaps both bytes and halfwords. Used only with the -i option.

-B Input/output is to be blocked 5,120 bytes to the record (does not apply to the -p option; meaningful only with data directed to or from 3.5-inch disks).

-c Writes *header* information in ASCII character form for portability.

-d Creates *directories* as needed.

directory
Specifies the directory to be copied.

-f Copies in all files except those in *patterns*.

-F Causes each floppy to be formatted after it is inserted into the drive, when used with the -o and when the output device is a Macintosh II floppy drive, the -F option. This formatting is for 800K drives only, so only 800K floppy disk should be used.

-i Specifies copy-in; extracts files from the standard input, which is assumed to be the product of a previous cpio -o. Only files with names that match *patterns* are selected. *patterns* are given in the name-generating notation of sh(1). In *patterns*, the meta-characters ?, *, and [...] match the slash / character. Multiple *patterns* may be specified but if none are, the default for *patterns* is * (that is, select all files). The extracted files are conditionally created and copied into the current directory tree based on the flag options described later. The permissions of the files will be those of the previous cpio -o. The owner and group assigned to the files will be that of the current user unless the user is the superuser, which causes cpio to retain the owner and group assigned to the files from the previous cpio -o. When cpio -i prints a message *xxx* blocks, it indicates how many blocks were read from the collection.

`-l` Links files (whenever possible) rather than copying them. Usable only with the `-p` option.

`-m` Retains previous file-modification time. This option is ineffective on directories that are being copied.

`-o` Specifies copy out; reads the standard input to obtain a list of pathnames and copies those files onto the standard output together with pathname and status information. The list of pathnames must contain only one file per line. (Thus, only certain commands, such as `find` or `ls` without the `-C` option, will work in a pipeline to `cpio`.) Output is padded to a 512-byte boundary. When `cpio -o` prints a message *xxx* `blocks`, it indicates how many blocks were written.

`-p` Specifies pass; reads the standard input to obtain a list of pathnames of files that are conditionally created and copied into the destination *directory* tree based on the flag options described later. When `cpio -p` prints a message *xxx* `blocks`, it indicates how many blocks were written.

patterns
Specifies a list of files that have special treatment.

`-r` Renames (interactively) files. If the user types a null line, the file is skipped.

`-s` Swaps bytes. Used only with the `-i` option.

`-S` Swaps halfwords. Used only with the `-i` option.

`-t` Prints a table of contents of the input. No files are created.

`-u` Copies unconditionally (normally, an older file will not replace a newer file with the same name).

`-v` Specifies verbose mode; causes a list of filenames to be printed. When used with the `-t` option, the table of contents looks like the output of an `ls -l` command (see `ls`(1)).

DESCRIPTION

`cpio` copies file and directories to or from a `cpio` archive. The `cpio` command does not follow symbolic links.

EXAMPLES

The pipeline:

```
ls | cpio -o > /dev/rdsk/c8d0s0
```

copies the contents of a directory into an archive.

The command:

```
cd olddir
find . -depth -print | cpio -pdl newdir
```

duplicates a directory hierarchy.

The simple case:

```
find . -depth -print | cpio -oB > /dev/rdsk/c8d0s0
```

may be handled more efficiently by:

```
find . -cpio /dev/rdsk/c8d0s0
```

LIMITATIONS

Pathnames are restricted to 128 characters.

If there are too many uniquely linked files, the program runs out of the memory needed to keep track of them and, thereafter, linking information is lost.

Only the superuser may copy special files.

FILES

`/bin/cpio`
Executable file

SEE ALSO

`ar`(1), `dd`(1), `find`(1), `ls`(1), `tar`(1)

`cpio`(4) in *A/UX Programmer's Reference*

NAME

cpp — invokes the C language preprocessor

SYNOPSIS

cpp [-C] [-D*name*[=*def*]] [-I*dir*] [-P] [-U*name*] [-M[*prefix*]] [-Y] [*ifile* [*ofile*]]

ARGUMENTS

-C Passes along all comments except those found on cpp directive lines. By default, cpp strips C-style comments.

-D*name*[=*def*]
Defines *name* as if by a #define directive. If no =*def* is given, *name* is defined as 1.

-I*dir*
Searches for #include files (whose names do not begin with /) in *dir* before looking in the directories on the standard list. When this option is used, #include files whose names are enclosed in " " (double quotes) are searched for first in the directory of the *ifile* argument, then in directories named in -I options, and last in directories on a standard list, which, at present, consists of /usr/include. If the -Y option (see below) is specified, the standard list is not searched. For #include files whose names are enclosed in <>, the directory of the *ifile* argument is not searched, unless -I. is specified.

-P Turns off the default production of line control information. Line control information is used by the next pass of the C compiler to generate useful error messages about the line on which an error occurred. Line control lines have the form

#lineno file

where *lineno* is the line number within the *file* at which cpp found the line. Line control information is useful because, as cpp reads each #include file and writes its contents to *ofile*, synchronization between the text lines in *ifile* and the text lines in *ofile* is lost. For example, if the following *ifile* is compiled without line control information

```
#include <stdio.h>

main()
{
        x =) 2;
}
```

the C compiler will generate the following error message:

```
"", line 157: x undefined
"", line 157: syntax error
```

To the C compiler, having included `stdio.h` in *ofile*, line 157 is the true line number at which the error occurred. With line control information enabled, which is the default, `cc` can display the correct line number as shown below:

```
"", line 5: x undefined
"", line 5: syntax error
```

`-U`*name*

Removes any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor. The list of reserved symbols is shown below:

operating system:
unix

hardware:
m68k

UNIX® System variant:
_SYSV_SOURCE
_BSD_SOURCE
_AUX_SOURCE

`-M`[*prefix*]

Generates `make` dependency statements from the `#include` (see below) statements contained in *ifile*. The dependency statements can later be incorporated into a description file for use by `make` in determining the `#include` files on which *ifile* depends. The dependency statements are of the form:

target: include1 [include2 ...]

where *target* is a name created by subsituting the suffix of *ifile* with '`.o`.' For example, if the name of *ifile* is `main.c`, *target* will be `main.o`. If *prefix* is present and the `#include` file in *ifile* is found in the standard list, *prefix* is applied as shown below:

target: prefix/include1 [prefix/include2 ...]

For example, if the source file `main.c` includes `<stdio.h>` and the `-M` option is used, the following dependency statement is generated:

```
main.o:     /usr/include/stdio.h
```

If `-M'$(INC)'` is specified, the following dependency statement is generated:

```
main.o:     $(INC)/stdio.h
```

Only unique #include files are kept; duplicates are discarded. The resulting *ofile* will contain only dependency statements. Error messages are written on standard error. If cpp is invoked with the -M option and *ifile* does not have a suffix, cpp exits with an error message.

-Y Prevents cpp from searching the standard list, which at present consists of /usr/include, when processing #include files. This option is useful when used in conjunction with the -I option. When cpp cannot find a #include file in the directories specified by one or more -I options, its normal behavior is to search the standard list. If the -Y option is used in addition to the -I option and cpp cannot find a #include file in the directory specified by one or more -I options, cpp writes an error message on standard error, does not search the standard list, and continues processing. Thus, use of the -Y option ensures that cpp does not silently include the wrong #include file in cases where the -I option is incorrectly specified or the desired #include file is erroneously missing from the specified directory.

ifile [*ofile*]
Specifies the input file to be compiled (*ifile*) and the output file (*ofile*) into which the results will be placed.

DESCRIPTION

cpp is the C language preprocessor that is invoked as the first pass of any C compilation using the cc(1) command. The output of cpp is acceptable as input to the next pass of the C compiler. As the C language evolves, cpp and the rest of the C compilation package will be modified to follow these changes. Therefore, the use of cpp other than in this framework is not suggested. The preferred way to invoke cpp is through the cc(1) command because the functionality of cpp may someday be moved elsewhere. See m4(1) for a general macro processor.

The cpp command optionally accepts two filenames as arguments. The input for the preprocessor is *ifile*, which is also known as the *source* file, and the output is *ofile*. If not supplied, *ifile* and *ofile* default to standard input and standard output, except in the case of the -M option, which requires *ifile* to be present and have a suffix.

Two special names are understood by cpp. The name __LINE__ is defined as the current line number (as a decimal integer) as known by cpp, and __FILE__ is defined as the current filename (as a C string) as known by cpp. They can be used anywhere, including in macros, just as any other defined name.

All cpp directives start with lines begun by #. The directives are:

#define *name* token-string
Replaces subsequent instances of *name* with *token-string*.

#define *name*(*arg*, ..., *arg*) token-string
Replaces subsequent instances of *name* followed by a left parenthesis [(], a list of comma-separated tokens, and a right parenthesis [)] by *token-string* where each occurrence of *arg* in the *token-string* is replaced by the corresponding token in the comma-separated list. Notice that there can be no space between *name* and the left parenthesis [(].

#undef *name*
Causes the definition of *name,* if any, to be undefined.

#include *"filename"*
#include *<filename>*
Includes at this point the contents of *filename,* which will then be run through cpp. When the *<filename>* notation is used, *filename* is only searched for in the standard places. See the -I option above for more detail.

#line *integer-constant "filename"*
Causes cpp to generate line-control information for the next pass of the C compiler. The line number of the next line is *integer-constant,* and *filename* is the file where it comes from. If "*filename*" is not given, the current filename is unchanged.

#endif
Ends a section of lines begun by a test directive (#if, #ifdef, or #ifndef). Each test directive must have a matching #endif.

#ifdef *name*
Displays the lines following in the output if and only if *name* was the subject of a previous #define without being the subject of an intervening #undef.

#ifndef *name*
Does not display the lines following in the output if and only if *name* has been the subject of a previous #define without being the subject of an intervening #undef.

#if *constant-expression*
Displays the lines following in the output if and only if *constant-expression* evaluates to nonzero. All binary nonassignment C operators, the ?: operator, the unary -, !, and ~ operators are legal in *constant-expression*. The precedence of the operators is the same as defined by the C language. There is also a unary operator

`defined` that can be used in *constant-expression* in these two forms: `defined( name )` or `defined` *name*. This allows the utility of `#ifdef` and `#ifndef` in an `#if` directive. Only these operators, integer constants, and names that are known by `cpp` should be used in *constant-expression*. In particular, the `sizeof` operator is not available.

`#elif` *constant-expression*
Displays the lines following the output if and only if *constant-expression* is true and the preceding `#if` *constant-expression* with which the subject `#elif` is paired is false. The formation of *constant-expression* is subject to the restrictions described for `#if` above.

`#else`
Reverses the notion of the test directive that matches this directive. If lines previous to this directive are ignored, the lines following appear in the output. If lines previous to this directive are not ignored, the lines following do not appear in the output.

`#ident ...`
This directive is maintained for historical reasons, but does not cause `cpp` to do any special processing nor to generate any output to *ofile*.

`#pragma ...`
Lines beginning with this directive are written to *ofile* without modification.

The test directives and the possible `#else` directives can be nested.

STATUS MESSAGES AND VALUES

The error and warning messages produced by `cpp` are self-explanatory and are written to standard output. The line number and filename where the error occurred are printed with the message. If *ifile* is a relative pathname, `cpp` also prints the absolute path of *ifile* in the form:

ifile (absolute path) :message

NOTES

When newline characters were found in argument lists for macros to be expanded, previous versions of `cpp` put out the newlines as they were found and expanded. The current version of `cpp` replaces these newlines with blanks to alleviate problems that the previous versions had when this occurred.

FILES

`/lib/cpp`
Executable file

`/usr/include`
Executable file

SEE ALSO

`cc`(1), `m4`(1), `make`(1)

"Other Programming Tools" in *A/UX Programming Languages and Tools, Volume 2*

NAME

crontab — aids in the use of the cron process scheduling program

SYNOPSIS

crontab [*file*]

crontab -l

crontab -r

ARGUMENTS

file Specifies the file to be copied.

-l Lists the crontab file for the invoking user.

-r Removes a user's crontab from the crontab directory.

DESCRIPTION

crontab is a utility that aids in the use of the cron process scheduling program. A crontab file stipulates the timetable for regular process scheduling. crontab copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. If standard input is used, an EOF (CONTROL-D by default in the A/UX standard distribution) must be entered to terminate the processes.

You are permitted to use crontab if your name appears in the file /usr/lib/cron/cron.allow. If that file does not exist, the file /usr/lib/cron/cron.deny is checked to determine if you should be denied access to crontab. If neither file exists, only root is allowed to submit a job. The allow/deny files consist of one username per line.

A crontab file consists of lines of six fields each. The fields are separated by spaces or tabs. The first five are integer patterns that specify the following:

minute (0-59),
hour (0-23),
day of the month (1-31),
month of the year (1-12),
day of the week (0-6with 0=Sunday).

Each of these patterns may be either an asterisk (meaning all legal values), or a list of elements separated by commas. An element is either a number, or two numbers separated by a minus sign (meaning an inclusive range). Note that the specification of days may be made by two fields (day of the month and day of the week). If both are specified as a list of elements, both are adhered to.

For example,

```
0 0 1,15 * 1
```

would run a command on the first and fifteenth of each month, as well as on every Monday. To specify days by only one field, the other field should be set to * (for example,

```
0 0 * * 1
```

would run a command only on Mondays). Thus, a secondary meaning of asterisk is ''use the other field.''

The sixth field of a line in a `crontab` file is a string that is executed by the shell at the specified times. A percent character in this field (unless escaped by \) is translated to a newline character. Only the first line (up to a % or end-of-line) of the command field is executed by the shell. The other strings following the percent character are made available to the command as standard input. `cron` reads only one line at a time. For example,

```
0 0 * * 1 cat %GO%HOME%EARLY%
```

would mail the output:

```
GO
HOME
EARLY
```

to the user at the requested time.

The shell is invoked from your `$HOME` directory with an `arg0` of `sh`. Users who desire to have their `.profile` executed must do so explicitly in the `crontab` file. `cron` supplies a default environment for every shell, defining:

```
HOME
LOGNAME
SHELL(=/bin/sh)
PATH(=:/bin:/usr/bin:/usr/lbin)
```

Note: Users should remember to redirect the standard output and standard error of their commands! If this is not done, any generated output or errors will be mailed to the user (via `mail`(1)).

FILES

`/usr/bin/crontab`
Executable file

`/usr/lib/crontab`
Executable file

`/usr/lib/cron`
Executable file

`/usr/spool/cron/crontabs`
Executable file

/usr/lib/cron/log
Executable file
/usr/lib/cron/cron.allow
Executable file
/usr/lib/cron/cron.deny
Executable file

SEE ALSO

at(1), sh(1)

cron(1M) in *A/UX System Administrator's Reference*

NAME

crypt — encodes and decodes passwords

SYNOPSIS

crypt [*password*]

ARGUMENTS

password

Specifies the password to be encoded or decoded.

DESCRIPTION

crypt reads from the standard input and writes on the standard output. The *password* is a key that selects a particular transformation. If no *password* is given, crypt demands a key from the terminal and turns off printing while the key is being typed in. The crypt command encrypts and decrypts with the same key:

```
crypt key <clear > cypher
crypt key < cypher| pr
```

will print the clear text file, clear.

Files encrypted by crypt are compatible with those treated by both the ed and ex editors in encryption mode.

The security of encrypted files depends on three factors: the fundamental method must be hard to solve; direct search of the key space must be infeasible; *sneak paths* by which keys or clear text can become visible must be minimized. The security of this scheme should not be relied on, for reasons described herein.

The crypt command implements a one-rotor machine designed along the lines of the German Enigma, but with a 256-element rotor. Methods of attack on such machines are known, but not widely; moreover, the amount of work required is likely to be large.

The transformation of a key into the internal settings of the machine is deliberately designed to be expensive, i.e., to take a substantial fraction of a second to compute. If keys are restricted to (for example) three lowercase letters, however, encrypted files may be read by expending only a substantial fraction of five minutes of machine time.

Since the key is an argument to the crypt command, it is potentially visible to users executing ps(1) or a derivative. To minimize this possibility, crypt takes care to destroy any record of the key immediately upon entry. The choice of keys and key security are the most vulnerable aspect of crypt.

EXAMPLES

The command:

```
crypt asa < sleeper.c > zzz
```

will use the string `asa` as key to the encryption algorithm to encrypt the contents of `sleeper.c`, and place the encrypted output in file `zzz`. The file `zzz` at this point will be unreadable. Note that the original file, `sleeper.c`, remains in readable form. To obtain a readable printout of the file `zzz`, it could be decoded as follows:

```
crypt < zzz
```

After the response:

```
Enter key:
```

the user types in: `asa`.

LIMITATIONS

If output is piped to `nroff` and the encryption key is not given on the command line, `crypt` may leave terminal modes in a strange state (see `stty`(1)).

If two or more files encrypted with the same key are concatenated and an attempt is made to decrypt the result, only the contents of the first of the original files will be decrypted correctly.

NOTES

This utility is not provided with international distributions.

FILES

`/bin/crypt`
Executable file

`/dev/tty`
Terminal device file

SEE ALSO

`ed`(1), `ex`(1), `makekey`(1), `stty`(1), `vi`(1)

`crypt`(3C) in *A/UX Programmer's Reference*

NAME

csh — runs the C shell, a command interpreter with C-like syntax

SYNOPSIS

csh [-c] [-e] [-f] [-i] [-n] [-s] [-t] [-v] [-V] [-x] [-X] [*arg*]...

ARGUMENTS

arg Specifies the program to run through the shell.

-c Reads commands from the (single) following argument, which must be present. Any remaining arguments are placed in argv.

-e Exits the shell if any invoked command terminates abnormally or yields a nonzero exit status.

-f Starts the shell faster because it will neither search for nor execute commands from the file .cshrc in the invoker's home directory.

-i Causes the shell to be 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 Takes the command input from the standard input.

-t Reads a single line of input and executes it. 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.

-V Causes the verbose variable to be set even before .cshrc is executed.

-x Causes the echo variable to be set, so that commands are echoed immediately before execution.

-X Causes the echo variable to be set even before .cshrc is executed.

DESCRIPTION

csh is a command language interpreter incorporating a history mechanism (see ''History Substitutions''), job control facilities (see ''Jobs''), and a C-like syntax. In order to use its job control facilities, users of csh must enable the generation of suspend characters with stty(1).

An instance of `csh` begins by executing commands from the file `.cshrc` in the home directory of the invoker. If this is a login shell, then it also executes commands from the file `.login` (also in the home directory). It is typical for users on CRT's to put the `tset`(1) command in their `.login` file.

In the normal case, the shell will then 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 then 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 then parsed. Finally each command in the current line is executed.

When a login shell terminates, it executes commands from the file `.logout` in the user's home directory.

Lexical Structure

The shell splits input lines into words at blanks and tabs, with the following exceptions. The characters `&`, `|`, `;`, `<`, `>`, `(`, and `)` 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 blank.

In addition 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 that continues to the end of the input line. It is prevented this special meaning when preceded by `\` and when using `'`, `'`, and `"` quotation.

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 placed in `()` to form a simple command (which may be a component of a pipeline, and so forth). 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 that looks like

```
[1] 1234
```

indicating that the job that 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`(2) it. This can usefully be typed ahead when you have prepared some commands for a job that you wish to stop after the program 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 as 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 later), %% is also a synonym for the current job.

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.

Substitutions

In this section, various transformations that the shell performs on the input are described, in the order in which they occur.

History Substitutions

History substitutions place words from previous command input in 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 a \ 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 that 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 that 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 this list 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 zach
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 `!?zach?` 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 the command) word being 0, the second word (first argument) being 1, and so forth. The basic word designators are

`0`	first (command) word
n	*n*th argument
`^`	first argument, that is, `1`
`$`	last argument
`%`	word matched by (immediately preceding) `?`*s*`?` search
x-*y*	range of words
-*y*	abbreviates `0-`*y*
`*`	abbreviates `^-$`, or nothing if only 1 word in event
x`*`	abbreviates *x*`-$`
x-	like *x*`*` but omitting word `$`

The `:` 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 `:`. 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, for example g&.
p	Print the new command 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 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 & on the right side is replaced by the text from the left. A \ quotes & also. A null *l* uses the previous string either from an *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, for example !$. In this case the reference is to the previous command unless a previous history reference occurred on the same line, in which case this form repeats the previous reference. Thus !?foo?^ !$ 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^, providing a convenient shorthand for substitutions on the text of the previous line. Thus ^lb^lib fixes the spelling of lib in the previous command. Finally, a history substitution may be surrounded with { and } if necessary to insulate it from the characters that follow. Thus, after ls -ld ~paul we might do !{l}a to do ls -ld ~paula, while !la would look for a command starting 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 prevented any further interpretation. Strings enclosed in " may be expanded as described later.

In both cases the resulting text becomes (all or part of) a single word; only in one special case (see ''Command Substitution'') 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 that 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 for `lookup` was `grep !^ /etc/passwd`, then `lookup tim` would map to `grep tim /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 say

```
alias print 'pr \!* | lpr '
```

to make a command that uses `pr` to send its arguments 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 or referred to by the shell. 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 that causes command input to be echoed. The setting of this variable results from the `-v` 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 double quotes where it *always* occurs, and within single quotes where it *never* occurs. Strings quoted by ` ‘ ` are interpreted later (see ‘‘Command Substitution’’), 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 at 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 that 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 that would otherwise be part of it. Shell variables have names consisting of up to 18 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 later 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 last member of a range is omitted it defaults to $#*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 *name*. This is useful for later use in a [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 :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.

Note: 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.

$$ Substitutes 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 built-in commands. This means that portions of expressions that are not evaluated are not subjected to these expansions. For commands that 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 `"`, 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 filenames that match the pattern. In a list of words specifying filename substitution, it is an error if no pattern matches an existing filename, but it is not required that each pattern 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, that is, as `~`, 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 the home directory; thus `~paul` might expand to `/usr/paul` and `~paul/chmach` to `/usr/paul/chmach`. If the character `~` is followed by a character other than a letter or `/` appears, but not at the beginning of a word, it is left undisturbed.

The metanotation `a{b,c,d}e` is 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 that 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 that are substituted have all blanks, tabs, and newlines preserved, except for the final newline that 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, it is truncated and its previous contents are lost. If the variable `noclobber` is set, then either the file must not exist or be a character special file (for example, 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 as well as the standard output into the specified file. *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 that 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.")

Diagnostic output may be directed through a pipe with the standard output. Simply use the form |& rather than just |.

Expressions

A number of the built-in 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 with those on the same line having equal precedence:

```
== != =~ !~
<= >= < >
<< >>
+ -
* / %
```

The ==, !=, =~, and !~ operators compare their arguments as strings; all others operate on numbers. The operators =~ and !~ are like == and !=

except that the right side is a pattern (containing, for example, `*`, `?`, and instances of `[. . .]`) against which the left 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 that 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 that 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, that is `0`. Command executions succeed, returning true, that is `1`, if the command exits with status 0; otherwise they fail, returning false, that is `0`. If more detailed status information is required then the command should be executed outside of an expression and the variable `status` examined.

Control Flow

The shell contains a number of commands that 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 later.

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, backward `goto`'s will succeed on nonseekable inputs.)

Built-in Commands

Built-in commands are executed within the shell. If a built-in 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 core in use, broken down into used and free core, and the address of the last location in the heap. `alloc` used with an argument shows each used and free block on the internal dynamic memory chain indicating its address, size, and whether it is used or free. This is a debugging command and may not work in production versions of the system 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`. The remaining commands on the current line are executed. Multilevel 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 later under `switch`.

`cd`
`cd` *name*
`chdir`
`chdir` *name*
: Changes 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 *name*. Finally, if all else fails but *name* is a shell variable whose value begins with `/`, then its value is tried to see if it is a directory.

`continue`
Continues execution of the nearest enclosing `while` or `foreach`. The rest of the commands on the current line are executed.

`default:`
Labels the default case in a `switch` statement. The default 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 shell's 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 later in this section.

`eval` *arg...*
The arguments are read as input to the shell (as in `sh`(1)) and the resulting command(s) is 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 this command and the matching `end` are executed. (Both `foreach` and `end` must appear singly on separate lines.)

The built-in command `continue` may be used to continue the loop prematurely and the built-in 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 interrupt it.

`glob` *wordlist*
Like `echo` but no `\` escapes are recognized and words are delimited by null characters in the output. Useful for programs that 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 of the form *label*. The shell rewinds its input as much as possible and searches for a line of the form *label*`:` possibly preceded by blanks or tabs. Execution continues after the specified line.

`hashstat`
Prints a statistics line indicating how effective the internal hash table has been at locating commands (and avoiding any `exec`). An `exec` is attempted for each component of the *path* where the hash function indicates a possible hit, and in each component that 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 and is used to produce files suitable for sourcing using the `-h` option to `source`.

`if (`*expr*`)` *command*
: If the specified *expr* evaluates true, then the single *command* with arguments is executed. In the interactive shell, the `if` statement can only accept one simple command after the *expr* and in the same line as *expr*. Variable substitution on *command* happens early, at the same time it does for the rest of the `if` command. The *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`
...
`else if (`*expr2*`) then`
...
`else`
...
`endif`
: If the specified *expr* is true, then the first command is executed. In the interactive shell, the `if then` statement can only accept one simple command after `then`. This command must be specified on the same line as `then`. If the specified *expr2* is true, then the command to the `else` or `else if` are executed, and so forth. 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; the `-l` option lists process ID's in addition to the normal information.

`kill %`*job*
`kill -`*sig* `%`*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; just saying `kill` does not send a signal to the current job. A *pid* of 0 means the current process (that is, this invocation of the C shell). Consequently, `kill -9 0` terminates the current C shell and possibly logs you off. 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.

`login`
Terminates 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`
Terminates a login shell. Especially useful if `ignoreeof` is set.

`nice`
`nice` *+number*
`nice` *command*
`nice` *+number command*
The first form sets the `nice` for this shell to 4. The second form sets the `nice` to the given number. The final two forms run command at priority 4 and `number` respectively. The superuser may specify negative niceness by using

`nice` *-number ...*

Command is always executed in a subshell, 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 run effectively without hangups.

`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*
Controls the action of the shell on interrupts. The first form restores the default action of the shell on interrupts that are 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 the argument *+n,* `popd` 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 (like `cd`) and pushes the old current working directory (as in `csw`) onto the directory stack. With a numeric argument, `pushd` rotates the *n*th argument of the directory stack around to be the top element and changes into 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 value of all shell variables. Variables that 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* 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` *name value*
Sets the value of the environment variable *name* to be *value*, a single 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 shell reads commands from *name*. `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. Normally, input during `source` commands is not placed on the history list; the `-h` option causes the commands to be placed in the history list without being executed.

`stop %`*job* ...
Stops the current or specified job that 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 str1:
...
    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, that

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 there to be nothing to `unalias`.

`unhash`
Use of the internal hash table to speed location of executed programs is disabled.

`unset` *pattern*
All variables whose names match the specified pattern are removed. Thus all variables are removed by `unset *`, which 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 and `printenv`(1).

`wait`
All background jobs are waited for. If 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 nonzero, 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 `()`. The third form assigns the value of *expr* to the *index* argument of *name*. Both *name* and its *index* component must already exist.

The operators *=, +=, and so forth, 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* that would otherwise be single words.

Special postfix ++ and -- operators increment and decrement *name* respectively; for example `@ i++`.

Predefined 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 then will not be modified except 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 change it.

`argv`	Set to the arguments to the shell, it is from this variable that positional parameters are substituted; that is, `$1` is replaced by `$argv[1]`, and so forth.
`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. `echo` causes each command and its arguments to be echoed just before it is executed. For nonbuilt-in commands, all expansions occur before echoing. Built-in commands are echoed before command and filename substitution, since these substitutions are then done selectively.
`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 that has been referenced in this many events will not be discarded. Values of `history` that are too large 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 that are terminals. This prevents shells from accidentally being killed by CONTROL-D's.
`mail`	The files where the shell checks for mail. This is done after each command completion that 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 modification 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 ensure 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 that 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 if a filename expansion does not match any existing files; rather the primitive pattern is returned. It is still an error for the primitive pattern to be malformed, that is, `echo [` still gives an error.

`notify` If set, the shell notifies asynchronously of job completions. The default is to present job completions just before printing a 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 pathnames will execute. The usual search path is `.`, `/bin`, and `/usr/bin`, but this may vary from system to system. For the superuser, the default search path is `/etc`, `/bin`, and `/usr/bin`. A shell that 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 give the `rehash` or the commands may not be found.

`prompt` The string that is printed before each command is read from an interactive terminal input. If an `!` 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` a numeric value is given to control the number of entries of the history list that are saved in `~/.history` when

	the user logs out. Any command that has been referenced in that number of events will be saved. During start up the shell sources `~/.history` into the history list, enabling history to be saved across logins. Values of `savehist` that are too large 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 that have execute bits set, but which are not executable by the system. (See the description of ''Nonbuilt-in Command Execution'' later.) 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. Built-in commands that fail return exit status `1`; all other built-in commands set status `0`.
`time`	Controls automatic timing of commands. If set, then any command that takes more than this many cpu seconds will cause a line to be printed when it terminates. This line shows user, system, and real times and a utilization percentage that is the ratio of user plus system times to real time
`verbose`	Set by the `-v` option. Causes the words of each command to be printed after history substitution.

Nonbuilt-in Command Execution

When a command to be executed is found to not be a built-in 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` argument (and in any case for each directory component of `path` that does not begin with a `/`), the shell concatenates with the given command name to form a pathname 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 the *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 prefixed to the argument list to form the shell command. The first word of the `alias` should be the full pathname of the shell (for example `$shell`). Note that this is a special, late occurring, case of `alias` substitution, and only allows words to be prefixed to the argument list without modification.

Argument List Processing

If argument 0 to the shell is `-` then this is a login shell.

After processing of 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 the standard Bourne shell (`/bin/sh`), 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 `#`; that is, if the script does not start with a comment. Remaining arguments initialize the variable `argv`.

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 that the shell inherited from its parent. The shells 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`.

LIMITATIONS

Words can be no longer than 1024 characters. The system limits argument lists to 10240 characters. The number of arguments to a command that involves filename expansion is limited to 1/6th 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.

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 (that is, wrong) as the job may have changed directories internally.

Shell built-in functions are not stoppable/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 execute *c*. This is especially noticeable if this expansion results from an `alias`. It suffices to place the sequence of commands in `()` to force it to a subshell.

(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 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 structure should be parsed rather than recognized as built-in commands, allowing 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.

Symbolic links fool the shell. In particular, `dirs` and

```
cd ..
```

don't work properly once you've crossed through a symbolic link.

FILES

`/bin/csh`
: Executable file

`~/.cshrc`
: File that is read at the beginning of execution by each shell

`/etc/cshrc`
: Global file that is read by the login shell before `~/.cshrc`

`~/.login`
: File that is read by the login shell after `.cshrc` at login.

`~/.logout`
: File that is read by the login shell, at logout

`/bin/sh`
: Standard shell file, for shell scripts not starting with a `#`.

`/tmp/sh*`
: Temporary file for `<<`.

`/etc/passwd`
Source of home directories for ~*name*.

SEE ALSO

`ksh`(1), `sh`(1)

`access`(2), `exec`(2), `fork`(2), `pipe`(2), `sigvec`(2), `umask`(2), `wait`(2), `killpg`(3N), `a.out`(4), `environ`(5), `tty`(7) in *A/UX Programmer's Reference*

"C Shell Reference" in *A/UX Shells and Shell Programming*

NAME

csplit — splits files into sections

SYNOPSIS

csplit [-f *prefix*] [-k] [-s] *file arg1* [... *argn*]

ARGUMENTS

arg1 [... *argn*]
Specifies the sections that the file is split into. Replace *arg1* with the first argument, and replace *argn* with the last argument.

-f *prefix*
Causes the created files to be named *prefix*00... *prefixn*. The default is xx00... *xx*n.

file Specifies the file to be split. If the *file* argument is a - then standard input is used.

-k Leaves previously created files intact. Normally removes created files if an error occurs.

-s Suppresses the printing of all character counts. Normally prints the character counts for each file created.

DESCRIPTION

csplit reads *file* and separates it into *n*+1 sections, defined by the arguments *arg1*... *argn*. By default, the sections are placed in files named xx00... xx*n* (*n* may not be greater than 99). These sections get the following pieces of *file*:

00: From the start of *file* up to (but not including) the line referenced by *arg1*.

01: From the line referenced by *arg1* up to the line referenced by *arg2*.

.

.

.

n+1: From the line referenced by *argn* to the end of *file*.

The arguments (*arg1... argn*) to `csplit` can be a combination of the following:

/rexp/
: A file is to be created for the section from the current line up to (but not including) the line containing the regular expression *rexp*. The current line becomes the line containing *rexp*. This argument may be followed by an optional + or - some number of lines (e.g., `/Page/-5`).

%rexp%
: This argument is the same as */rexp/*, except that no file is created for the section.

lnno
: A file is to be created from the current line up to (but not including) *lnno*. The current line becomes *lnno*.

{*num*}
: Repeat argument. This argument may follow any of the above arguments. If it follows a *rexp* type argument, that argument is applied *num* more times. If it follows *lnno*, the file will be split every *lnno* lines (*num* times) from that point.

Enclose all *rexp* type arguments that contain blanks or other characters meaningful to the shell in the appropriate quotes. Regular expressions may not contain embedded newlines. `csplit` does not affect the original file; it is the user's responsibility to remove it.

EXAMPLES

The command:

```
csplit -f cobol file '/procedure
division/' /par5./  /par16./
```

creates four files, `cobol00 ... cobol03`. After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

```
csplit -k file  100  {99}
```

splits the file at every 100 lines, up to 10,000 lines. The `-k` option causes the created files to be retained if there are less than 10,000 lines; however, an error message would still be printed.

```
csplit -k prog.c  '%main(%'  '/^}/+1'  {20}
```

assuming that `prog.c` follows the normal `C` coding convention of ending routines with a `}` at the beginning of the line, this example will create a file

containing each separate C routine (up to 21) in prog.c.

STATUS MESSAGES AND VALUES

Self explanatory except for:

```
arg - out of range
```

which means that the given argument did not reference a line between the current position and the end of the file.

FILES

/usr/bin/csplit
Executable file

SEE ALSO

ed(1), fsplit(1), sh(1), split(1)

regexp(5) in *A/UX Programmer's Reference*

NAME

ct — runs login on a dial-up line

SYNOPSIS

ct [-c*device-type*] [-h] [-l*device-name*] [-s*baud-rate*] [-v] [-w*time-limit*] [-x*debug-level*] *telephone-number ...*

ARGUMENTS

-c*device-type*
: Causes ct to use only those entries in the Devices file whose *device-type* field matches the specified value. If you use this option, you must also supply a device name by using the -l option.

-h
: Prevents ct from hanging up the current line. If you use this option, you cannot enter another command until ct is done.

-l*device-name*
: Specifies the name of the device to use for establishing the connection. If you do not use the -s option with the -l option, ct uses the baud rate of the first entry in the Devices file whose *device-name* field matches the value of *device-name*. If the baud rate of the selected entry is Any, the default baud rate, as specified by the Default_Baudrate keyword in the Config file, is used. If you use both the -l and the -s options, ct searches the Devices file to verify that specified baud rate is available.

-s*baud-rate*
: Sets the baud rate. The default is 2400.

telephone-number
: Specifies the telephone number to dial. You can use numbers from 0 to 9, the minus sign (-), the equal sign (=), the asterisk (*), and the number sign (#). Use an equal sign to wait for a secondary dial tone and a minus sign to pause. The maximum length of *telephone-number* is 31 characters. If you specify more than one telephone number, ct tries each number in succession until one answers.

-v
: Specifies verbose mode. If you use this option, ct writes a log of its actions on the standard error.

-w*time-limit*
: Specifies in minutes the time to wait for a free dialer before giving up. The value of *time-limit* must be greater than 0.

-x*debug-level*
: Causes ct to write a detailed account of its actions on the standard error. The value of *debug-level* must be a number from 0 to 9. Higher numbers produce more detailed debugging information.

DESCRIPTION

`ct` dials a telephone number and runs `login` on the line if a connection is established. The connection can be to a terminal or another computer. You can use `cron` to run `ct` at a specified time, in which case, be sure to use the `-h` option. If `getty` is set up on the remote A/UX system to handle both dial-in and dial-out connections, you can dial in to that system and run `ct`, which drops your dial-in connection and calls you back.

EXAMPLES

Here is an example of using `ct` when you have used a dial-in connection to log in to an A/UX system whose `getty` is set up to handle both dial-in and dial-out connections. First, run the `tty` command to find out the name of the device that is being used for your current connection. Then run `ct`.

Here is a command that runs `ct` on `tty0` at 2400 baud:

```
ct -ltty0 -s2400 5551212
```

The `ct` command then displays this message:

```
Allocated ACU dialer device-name [this line] at baud-rate baud
Confirm hangup? (y/n)
```

If you enter `n`, `ct` exits and your prompt is returned. If you enter `y`, `ct` drops the connection and calls you back, using the telephone number you supplied on the `ct` command line, and presents the login prompt. When you are ready to end the connection, log out; `ct` issues this prompt:

```
Reconnect?
```

If you want to reconnect, enter `yes`. If you want to disconnect, enter `no`.

FILES

`/bin/ct`
: Executable file

`/usr/lib/uucp/Config`
: File that specifies the value of UUCP configuration parameters

`/usr/lib/uucp/Devices`
: File that describes the devices that can be used

`/usr/spool/uucp/.Admin/ctlog`
: File that logs the use of `ct` on your system

SEE ALSO

`login`(1)

`getty`(1M) in *A/UX System Administrator's Reference*

Chapter 8, "Setting Up the UUCP System," in *A/UX Network System Administration*

NAME

ctags — maintains a tags file for a C program

SYNOPSIS

ctags [-a] [-u] [-w] [-x] *file...*

ARGUMENTS

-a Causes the output to be appended to the tags file instead of rewriting it.

file Specifies the C program file to be tagged.

-u Causes the specified files to be updated in tags; that is, all references to them are replaced by new values. (Beware: this option is implemented in a way that is rather slow; it is usually faster to simply rebuild the tags file.)

-w Suppresses warning diagnostics.

-x Produces a list of function names, the line number and file name on which each is defined, as well as the text of that line and prints this on the standard output.

DESCRIPTION

ctags makes a tags file for ex(1) and vi(1) from the specified C, Fortran, and Pascal sources.

A tags file gives the locations of specified objects (in this case functions) in a group of files. Each line of the tags file contains the function name, the file in which it is defined, and a scanning pattern used to find the function definition. These are given in separate fields on the line, separated by blanks or tabs. Using the tags file, ex can quickly find these function definitions.

Files whose name ends in .c or .h are assumed to be C source files and are searched for C routine and macro definitions.

The tag *main* is treated specially in C programs. The tag formed is created by prefixing M to the name of the file, with a trailing .c removed, if any, and leading pathname components also removed. This makes use of ctags practical in directories with more than one program.

EXAMPLES

The command:

```
ctags *.c *.h
```

puts the tags from all the .c and .h files into the tagsfile tags.

LIMITATIONS

Not all warning diagnostics are suppressed by the `-w` option.

If `ctags`(1) is interrupted while executing under the `-u` option, a temporary file named `OTAGS` is left in the current directory.

FILES

`/usr/bin/ctags`
: Executable file

`tags`
: Output tags file

SEE ALSO

`ex`(1), `vi`(1)

NAME

ctrace — debugs a C program

SYNOPSIS

ctrace [-b] [-e] [-f*functions*] [-l*n*] [-o] [-p *'s'*] [-P] [-r*f*] [-s] [-t*n*] [-u] [-v*functions*] [-x] [*file*]

ARGUMENTS

-b Uses only basic functions in the trace code, that is, those in ctype(3C), printf(3S), and string(3C). These are usually available even in cross-compilers for microprocessors. In particular, this option is needed when the traced program runs under an operating system that does not have signal(3), fflush(3S), longjmp(3C), or setjmp(3C).

-e Specifies the floating point format.

-f*functions*
Traces only these *functions*.

-l*n*
Checks *n* consecutively executed statements for looping trace output, instead of the default of 20. Use 0 to get all the trace output from loops.

-o Specifies the octal format.

-p *'s'*
Changes the trace print function from the default of 'printf('. For example, 'fprintf(stderr' would send the trace to the standard error output.

-P Runs the C preprocessor on the input before tracing it. You can also use the -D, -I, and -U cc(1) preprocessor options.

-r*f* Uses file *f* in place of the runtime.c trace function package. This lets you change the entire print function, instead of just the name and leading arguments (see the -p option).

-s Suppresses redundant trace output from simple assignment statements and string copy function calls. This option can hide a bug caused by use of the = operator in place of the == operator.

-t*n*
Traces *n* variables per statement instead of the default of 10 (the maximum number is 20). The ''Status Messages and Values'' section explains when to use this option.

-u Specifies the unsigned format.

-v*functions*
Traces all but these

`-x` Specifies the hexadecimal format.

file Specifies the file to be debugged. If this argument is not given, `ctrace` will read the file from the standard input.

DESCRIPTION

`ctrace` allows you to follow the execution of a C program, statement by statement. The effect is similar to executing a shell procedure with the `-x` option. The `ctrace` command reads the C program in *file* (or from standard input if you do not specify *file*), inserts statements to print the text of each executable statement and the values of all variables referenced or modified, and writes the modified program to the standard output. You must put the output of `ctrace` into a temporary file because the `cc`(1) command does not allow the use of a pipe. You then compile and execute this file.

As each statement in the program executes, it will be listed at the terminal, followed by the name and value of any variables referenced or modified in the statement, followed by any output from the statement. Loops in the trace output are detected and tracing is stopped until the loop is exited or a different sequence of statements within the loop is executed. A warning message is printed every 1000 times through the loop to help you detect infinite loops. The trace output goes to the standard output, so that you may put it into a file for examination with an editor or the `bfs`(1) or `tail`(1) commands.

You may want to print variables in other formats besides the default. Long and pointer variables are always printed as signed integers. Pointers to character arrays are also printed as strings if appropriate. `char`, `short`, and `int` variables are also printed as signed integers and, if appropriate, as characters. Double variables are printed as floating point numbers in scientific notation.

EXAMPLES

If the file `lc.c` contains this C program:

```
 1 #include <stdio.h>
 2 main()   /* count lines in input */
 3 {
 4    int c, nl;
 5
 6    nl = 0;
 7    while ((c = getchar()) != EOF)
 8          if (c = '\n')
 9                ++nl;
10    printf("%d\n", nl);
11 }
```

and you enter these commands and test data:

```
cc lc.c
a.out
1
```

the program will be compiled and executed. The output of the program will be the number 2, which is not correct because there is only one line in the test data. The error in this program is common, but subtle. If you invoke `ctrace` with these commands:

```
ctrace lc.c > temp.c
cc temp.c
a.out
```

the output will be:

```
 2 main()
 6     nl = 0;
       /* nl == 0 */
 7     while ((c = getchar()) != EOF)
```

The program is now waiting for input. If you enter the same test data as before, the output will be:

```
       /* c == 49 or '1' */
 8           if (c = '\n')
             /* c == 10 or '\n' */
 9                 ++nl;
                   /* nl == 1 */
 7     while ((c = getchar()) != EOF)
       /* c == 10 or '\n' */
 8           if (c = '\n')
             /* c == 10 or '\n' */
 9                 ++nl;
                   /* nl == 2 */

 7                 /* repeating */
```

If you now enter an end-of-file character (CONTROL-d), the final output will be:

```
       /* c == -1 */
10     printf("%d\n", nl);
       /* nl == 2 */2
       /* return */
```

Note that the program output printed at the end of the trace line for the `nl` variable. Also note the `return` comment added by `ctrace` at the end of

the trace output. This shows the implicit return at the terminating brace in the function.

The trace output shows that variable `c` is assigned the value "`1`" in line 7, but in line 8 it has the value "`\n`". Once your attention is drawn to this `if` statement, you will probably realize that you used the assignment operator (=) in place of the equal operator (==). During code reading, it is easy to miss this error.

Execution-time Trace Control

The default operation for `ctrace` is to trace the entire program file, unless you use the `-f` or `-v` options to trace specific functions. This does not give you statement by statement control of the tracing, nor does it let you turn the tracing off and on when executing the traced program.

You can do both of these by adding `ctroff` and `ctron` function calls to your program to turn the tracing off and on, respectively, at execution time. Thus, you can code arbitrarily complex criteria for trace control with `if` statements, and you can even conditionally include this code because `ctrace` defines the `CTRACE` preprocessor variable. For example:

```
#ifdef CTRACE
      if (c == '!' && i > 1000)
            ctron();
#endif
```

You can also call these functions from `sdb`(1) if you compile with the `-g` option. For example, to trace all but lines 7 to 10 in the main function, enter:

```
sdb a.out
main:7b ctroff()
main:11b ctron()
r
```

You can also turn the trace off and on by setting the static variable `tr_ct_` to 0 and 1, respectively. This is useful if you are using a debugger that cannot call these functions directly.

STATUS MESSAGES AND VALUES

This section contains messages from both `ctrace` and `cc`(1), since the traced code often gets some `cc` warning messages. You can get `cc` error messages in some rare cases, all of which can be avoided.

`ctrace` Messages

`Warning: some variables are not traced`

Only 10 variables are traced in a statement to prevent the C compiler "out of tree space; simplify expression" error. Use the `-t` option to increase this number.

`Warning: statement too long to trace`
This statement is over 400 characters long. Make sure that you are using tabs to indent your code, not spaces.

`Cannot handle preprocessor code, use -P`
This is usually caused by `#ifdef/#endif` preprocessor statements in the middle of a C statement, or by a semicolon at the end of a `#define` preprocessor statement.

`''if ... else if'' sequence too`
Split the sequence by removing an `else` from the middle.

`Possible syntax error, try -P option`
Use the `-P` option to preprocess the `ctrace` input, along with any appropriate `-D`, `-I`, and `-U` preprocessor options. If you still get the error message, check the WARNINGS section below.

`cc` Messages

`Warning: floating point not implemented`
`Warning: illegal combination of pointer and`
`Warning: statement not reached`
`Warning: sizeof returns 0`
Ignore these messages.

`Compiler takes size of function`
See the `ctrace ''possible syntax error''` message above.

`yacc stack overflow`
See the `''ctrace 'if ... else if' sequence` message, above.

`Out of tree space; simplify expression`
Use the `-t` option to reduce the number of traced variables per statement from the default of 10. Ignore the `''ctrace: too many variables to trace''` warnings you will now get.

`redeclaration of signal`
Either correct this declaration of `signal`(3), or remove it and #include <signal.h>.

WARNINGS

You will get a `ctrace` syntax error if you omit the semicolon at the end of the last element declaration in a structure or union, just before the right brace (}). This is optional in some C compilers.

Defining a function with the same name as a system function may cause a syntax error if the number of arguments is changed. To fix this, just use a different name.

The ctrace command assumes that BADMAG is a preprocessor macro, and that EOF and NULL are #define d constants. Declaring any of these to be variables, e.g., int EOF will cause a syntax error.

LIMITATIONS

The ctrace command does not know about the components of aggregates like structures, unions, and arrays. It cannot choose a format to print all the components of an aggregate when an assignment is made to the entire aggregate. The ctrace command may choose to print the address of an aggregate or use the wrong format (e.g., %e for a structure with two integer members) when printing the value of an aggregate.

Pointer values are always treated as pointers to character strings.

The loop trace output elimination is done separately for each file of a multi-file program. This can result in functions called from a loop still being traced, or the elimination of trace output from one function in a file until another in the same file is called.

FILES

/usr/bin/ctrace
Executable file

/usr/lib/ctrace
Directory containing source files

SEE ALSO

sdb(1)

ctype(3C), fclose(3S), printf(3S), setjmp(3C), signal(3), string(3C) in *A/UX Programmer's Reference*

NAME

cu — establishes an interactive connection with another system

SYNOPSIS

cu [-b*bits*] [-dhint] [-e] [-c*device-type*] [-o] [-s*baud-rate*] [-x*debug-level*] -l*device-name*

cu [-b*bits*] [-dhint] [-e] [-c*device-type*] [-l*device-name*] [-o] [-s*baud-rate*] [-x*debug-level*] *telephone-number*

cu [-b*bits*] [-dhint] [-e] [-c*device-type*] [-l*device-name*] [-o] [-s*baud-rate*] [-x*debug-level*] *system*

ARGUMENTS

-b*bits*
: Sets the bits per character. The value of *bits* can be 7 or 8. This option allows communications between systems with different character sizes. The default value of *bits* is the current character size set for your local terminal. Use `stty -a` to see the character-size setting. This setting will be `cs7` or `cs8`. If you set the bits per character to 8, do not use the `-e` or `-o` option.

-c*device-type*
: Causes `cu` to use only those entries in the `Devices` file whose *device-type* field matches the value specified by *device-type*. If you use this option, you must also supply a *system* argument or use the `-l` option.

-d
: Causes a diagnostic trace to be displayed while the connection is open. This option is equivalent to using `-x9`.

-e
: Designates that even parity is to be generated for data sent to the remote system. If you use this option, the number of bits per character must be 7.

-h
: Sets half-duplex mode. This option emulates the local `echo` command in order to support calls to other computer systems that expect terminals to be set to half-duplex mode.

-i
: Causes `cu` to ignore modem control signals.

-l*device-name*
: Specifies the name of the device to use for establishing the connection. This option overrides the default method used by `cu` to determine the connection medium, as described in ''Connection Phase'' later in this manual page. This option is especially useful when a serial cable directly connects the remote system to your system. In this case, you do not need to provide a *system* argument.

If you do not use the `-s` option with the `-l` option, `cu` uses the baud rate of the first entry in the `Devices` file whose *device-name* field matches the value of *device-name*. If the baud rate of the selected entry is `Any`, the default baud rate, as specified by the `Default_Baudrate` keyword in the `Config` file, is used.

If you use both the `-l` and the `-s` options, `cu` searches the `Devices` file to verify that specified baud rate is available. If the verification succeeds, the connection is made at the requested baud rate; otherwise, an error message is printed and the connection is not made.

`-n` Causes `cu` to prompt for the *telephone-number* argument. You should use this option when you incorporate the `cu` command in a shell script if you do not want to incorporate the telephone number in the script for security reasons.

`-o` Designates that odd parity is to be generated for data sent to the remote system. If you use this option, the number of bits per character must be 7.

`-s`*baud-rate*
Specifies the baud rate; 300, 1200, 2400, 4800, and 9600 are common values. See `termio`(7) for a list of valid baud rates. The default value depends on the order of entries in the `Devices` file.

`-t` Causes `cu` to dial a terminal that has been set to answer automatically. Appropriate mapping of carriage returns to carriage-return-line-feed pairs is set.

system
Specifies the name of a system. When you supply a *system* argument, `cu` checks the `Systems` file for an entry whose first field matches the value of *system* and uses the third field of the matching entry to determine the connection medium. To override this method, use the `-c` option. To see a list of systems to which your computer is configured to connect, use the `uuname` command.

telephone-number
Specifies the telephone number to dial. The value of *telephone-number* can consist of the digits 0 to 9, the asterisk (`*`), the number sign (`#`), the equal sign (`=`), and the minus sign (`-`). Use the equal sign to wait for a secondary dial tone, and the minus sign to pause for approximately 4 seconds.

`-x`*debug-level*
Writes debugging information on standard output. The value of *debug-level* is a number from 0 to 9. Higher numbers produce more detailed debugging information.

DESCRIPTION

`cu` establishes an interactive connection with another A/UX system, another UNIX® system, or a non-UNIX system. Once connected, you can also use `cu` to transfer files.

You can think of `cu` as operating in two phases. The first phase is the connection phase, in which the connection is established. The second phase is the interactive phase, which `cu` enters when the connection is established.

Connection Phase

The `cu` command uses the `Devices` and `Systems` files, which are also used by the `uucico` command, to establish the connection to another system. The `ct` command also uses the `Devices` file. For `cu` to work, your system administrator must set up these files in view of the connection media that are available for your system.

The `Devices` file specifies the medium by which a connection is made to the remote system. The medium can be a direct serial connection or an asynchronous modem and a telephone line.

The `Systems` file contains information about how to connect to a remote system.

By default, `cu` assumes that if a *telephone-number* argument appears on the command line, it should examine the `Devices` file to find the first entry that begins with the keyword `ACU` and use the device that is associated with that entry if available. If the device is not available, `cu` searches the remainder of that file, looking for an `ACU` entry with an available device. You can override this default search mechanism by using the `-l` option.

By default, `cu` assumes that if a *system* argument appears on the command line, it should examine the `Systems` file for an entry whose first field matches the value of *system*. If `cu` finds a match, it uses the information in the entry to determine the medium (direct serial connection or autodialing modem) to use to make the connection. You can override this default search mechanism by using the `-l` option.

Conversation Phase

Once the connection is made, `cu` runs as two processes: a ''transmit'' process that reads data from the standard input and, except for lines beginning with a tilde (`~`), passes the data to the remote system and a ''receive'' process that accepts data from the remote system and, except for lines beginning with `~`, passes the data to the standard output. By default, `cu` uses CONTROL-S and CONTROL-Q to control input from the remote to prevent buffer overrun.

The cu transmit process interprets user-initiated lines that begin with ~ as special commands. Here are the available commands:

~. Terminates the connection.

~! Starts an interactive shell on the local system.

~!*cmd...*
Runs the commands specified by *cmd* on the local system, using sh -c. Note that ~!cd causes the command to be run by a subshell, which is probably not what was intended. See the description of ~%cd later in this list, for an alternative.

~$*cmd...*
Runs the command specified by *cmd* on the local system and sends its output to the remote system.

~%cd *directory*
Changes your current working directory on the local system.

~%take *from* [*to*]
Copies the file on the remote system specified by *from* to the file specified by *to* on the local system. If *to* is omitted, the name of the file on the remote system is used as the name of the copied file on the local system.

~%put *from* [*to*]
Copies the file on the local system specified by *from* to the file specified by *to* on the remote system. If *to* is omitted, the name of the file on the local system is used as the name of the copied file.

~~*line*
Sends the line ~*line* to the remote system.

~%bits *number*
Changes the number of bits per character to the value specified by *number*. The value of *number* can be 7 or 8.

~%break
Transmits a break character to the remote system. The short name of this command is ~%b.

~%debug
Toggles the -d option on and off. The short name of this command is ~%d.

~%even
Changes the parity to even parity.

~h Displays a list of the cu special commands.

~%none
Disables parity generation and checking.

~%odd
Changes the parity to odd parity.

~%speed*baud-rate*
Changes the baud rate to the value specified by *baud-rate*, such as, for example, 4800.

~t Prints the values of the termio structure for your terminal. This option is useful when you are having problems connecting to a certain computer or using a certain modem.

~l Prints the values of the termio structure for the communication line that cu is using. This option is useful when you are debugging.

~%ifc
Toggles the input flow-control setting. When this command is enabled, which is the default, the local system controls input from the remote system by using the CONTROL-S and CONTROL-Q flow-control protocol. The flow-control protocol is useful only if the remote system also uses the protocol. Another name for this command is ~%nostop.

~%ofc
Toggles the output flow-control setting. When this command is enabled, the remote system can control output from the local system by using the CONTROL-S and CONTROL-Q flow-control protocol. This option is useful only if the remote system also uses the protocol. Another name for this command is %noostop.

~%divert
Toggles the setting for allowing or disallowing diversions that are not specified by the ~%take command.

~%old
Toggles the setting for allowing or disallowing old-style syntax for received diversions.

The ''receive'' process normally copies data from the remote system to the standard output of the local system. It can also direct the output to local files.

The ~%put command requires the presence of the stty and cat commands on the remote side, which limits the use of ~%put to connections with other UNIX systems. The ~%put command also requires that the erase and cancel characters on the remote system be identical to these characters on the local system. See stty(1) for details. The ~%put command inserts backslashes at appropriate places.

The ~%take command requires the presence of echo and cat on the remote system, which limits the use of ~%take to connections with other UNIX systems. To copy files without expanding tabs to spaces, use the stty command on the remote system to turn on the tabs mode.

When you use cu on your local system X to connect to system Y and then use cu on system Y to connect to system Z, you can run commands on system Y by using ~~. To remind yourself of which system the command is being run on, use the uname command. For example, uname can be run on systems X, Y, and Z as follows:

```
uname
```

System Z replies with Z. You then type

```
~!uname
```

Your local system, X, replies with X. You then type

```
~~!uname
```

System Y replies with Y. In general, the absence of a tilde causes the command to be run on the system that you most recently logged in to. A single tilde causes the command to be run on your local system. Two or more tildes cause the command to be run on the next system in the chain.

EXAMPLES

This command dials a system whose telephone number is 9 1 201 555 1234 on a telephone system where a dial tone is expected after the 9, using a baud rate of 1200:

```
cu -s1200 9=12015551234
```

If you do not specify the -s option, cu uses the baud rate specified in the first entry in the Devices file that begins with ACU.

This command connects to a system by using a serial cable attached to /dev/tty0 on the local system:

```
cu  -l/dev/tty0
```

You can also specify the argument to the -l option as tty0.

This command specifies a baud rate and a device:

```
cu -s1200 -ltty00
```

This command connects to a system by using a specific device that is connected to an autodialing modem:

```
cu -l/dev/tty0  9=12015551234
```

On a Macintosh running A/UX, the argument to the -l option can also be /dev/modem or modem.

This command connects to a system named `walrus`, which, depending on the `walrus` entry in the `Systems` file, can be reached by a direct connection or an autodialing modem.

```
cu walrus
```

STATUS MESSAGES AND VALUES

To indicate a normal termination condition, `cu` exits with 0; to indicate an error condition, `cu` exits with 1.

LIMITATIONS

The `cu` command does not do any integrity checking on the data it transfers. See `uucp`(1C) and `rcp`(1C) for transfer programs that perform integrity checking.

Data containing special `cu` characters, such as a tilde, may not transmit properly.

The `~%put` and `~%take` commands do not dependably transmit nonprinting characters.

A `cu` connection between some modems does not display the login prompt immediately upon connection. Pressing RETURN causes the prompt to be displayed.

The `~%put` and `~%take` commands cannot be used over a chain of three or more systems. You must move the files one system at a time.

During file transmissions using the `~%put` command, `cu` artificially slows transmission to reduce the risk of data loss.

Any file transferred with `~%take` or `~%put` must contain a trailing newline character; otherwise, the transfer is suspended. Pressing CONTROL-D usually clears the condition.

FILES

`/usr/bin/cu`
: Executable file

`/usr/lib/uucp`
: Directory of `uucp` commands and configuration files

`/usr/lib/uucp/Devices`
: File containing configuration information for devices

`/usr/lib/uucp/`*file*
: File used exclusively by `cu` or `uucico`, as specified by the `Sysfiles` file, containing configuration information for remote systems

`/usr/lib/uucp/Sysfiles`
: Optional file specifying that `cu` or `uucico` is to use a file other than the default file, `Systems`, to get configuration information for remote

systems

`/usr/lib/uucp/Systems`
File containing configuration information for remote systems

`/usr/spool/locks/*`
Directory of lock files, which are used to indicate that a serial device is in use.

SEE ALSO

`cat`(1), `echo`(1), `stty`(1), `uname`(1), `uucp`(1C), `uuname`(1C)

Chapter 8, ''Setting Up the UUCP System,'' in *A/UX Network System Administration*

NAME

cut — cuts out selected fields of each line of a file

SYNOPSIS

cut -c*list* [-s] [*file*]...

cut -f*list* [-d *char*] [-s] [*file*]...

ARGUMENTS

-c*list*
: Specifies character positions (e.g., -c1-72 would pass the first 72 characters of each line). *list* is a comma-separated list of integer field numbers (in increasing order), with optional - to indicate ranges as in the -o option of nroff/troff for page ranges; e.g., 1,4,7; 1-3,8; -5,10 (short for 1-5,10); or 3- (short for third through last field).

-d *char*
: Specifies the field delimiter. Used with the -f option only. The default is tab. Space or other characters with special meaning to the shell must be quoted.

file
: Specifies the file to be affected. If no files are given, the standard input is used.

-f*list*
: Specifies 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 fields only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless the -s option is specified. *list* is a comma-separated list of integer field numbers (in increasing order), with optional - to indicate ranges as in the -o option of nroff/troff for page ranges; e.g., 1,4,7; 1-3,8; -5,10 (short for 1-5,10); or 3- (short for third through last field).

-s
: Suppresses lines with no delimiter characters in case of -f option. Unless specified, lines with no delimiters will be passed through untouched.

DESCRIPTION

cut cuts out columns from a table or fields from each line of a file; in database parlance, it implements the projection of a relation. The cut command may be used as a filter.

The fields as specified by *list* in the -c and -f options may be fixed length, i.e., character positions as on a punched card (-c option) or the length may vary from line to line and be marked with a field delimiter character like TAB (-f option).

Use grep(1) to make horizontal cuts (by context) through a file, or paste(1) to put files together column-wise (i.e., horizontally). To reorder columns in a table, use cut and paste.

EXAMPLES

Use the command for:

```
cut -d: -f1,5 /etc/passwd
```

mapping user IDs to names.

Use the command:

```
name='who am i | cut -f1 -d" "'
```

to set *name* to current login name.

STATUS MESSAGES AND VALUES

`Line too long`
A line can have no more than 1023 characters or fields.

`Bad list for c/f option`
Missing the `-c` or `-f` option or incorrectly specified *list*. No error occurs if a line has fewer fields than the *list* calls for.

`No fields`
The *list* is empty.

FILES

`/usr/bin/cut`
Executable file

SEE ALSO

awk(1), colrm(1), grep(1), paste(1), sed(1)

NAME

cw, checkcw — prepare constant-width text for otroff

SYNOPSIS

cw [-d] [-f*n*] [-l*xx*] [-r*xx*] [-t] [+t] [*file*]...

checkcw [-l*xx*] [-r*xx*] *file*...

ARGUMENTS

-d Prints current option settings on file descriptor 2 in the form of otroff(1) comment lines. This option is meant for debugging.

-f*n*
The CW font is mounted in font position *n*; acceptable values for *n* are 1, 2, and 3 (default is 3, replacing the bold font). This option is only useful at the beginning of a document.

-l*xx*
Specifies the one- or two-character string *xx* to be the left delimiter; if *xx* is omitted, the left delimiter becomes undefined, which it is initially. The left and right delimiters may (but need not) be different.

-r*xx*
Specifies the one- or two-character string *xx* to be the right delimiter; if *xx* is omitted, the right delimiter becomes undefined, which it is initially. The left and right delimiters may (but need not) be different.

-t Turns transparent mode *off*.

+t Turns transparent mode *on* (this is the initial default).

file Specifies the file to be processed. The cw command reads the standard input when no *files* are specified (or when - is specified as the last argument), so it can be used as a filter.

DESCRIPTION

cw is a preprocessor for otroff(1) input files that contain text to be typeset in the constant-width (CW) font.

Text typeset with the CW font resembles the output of terminals and of line printers. This font is used to typeset examples of programs and of computer output in user manuals, programming texts, etc. (An earlier version of this font was used in typesetting *The C Programming Language* by B. W. Kernighan and D. M. Ritchie.) It has been designed to be quite distinctive (but not overly obtrusive) when used together with the Times Roman font.

Because the CW font contains a *nonstandard* set of characters and because text typeset with it requires different character and interword spacing than is used for *standard* fonts, documents that use the CW font must be preprocessed by `cw`.

The CW font contains the 94 printing ASCII characters:

```
abcdefghijklmnopqrstuvwxyz
ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789
!$%&()''*+@.,/:;=?[]|-_^~"<>{}#\
```

plus nine non-ASCII characters represented by four-character `otroff`(1) names (in some cases attaching these names to nonstandard graphics):

character	symbol	`otroff` name
Cents sign	¢	\(ct
EBCDIC *not* sign	¬	\(no
Left arrow	←	\(<-
Right arrow	→	\(->
Down arrow	↓	\(da
Vertical single quote	′	\(fm
Control-shift indicator	†	\(dg
Visible space indicator	□	\(sq
Hyphen	-	\(hy

The hyphen is a synonym for the unadorned minus sign (-). Certain versions of `cw` recognize two additional names: `\(ua` for an up arrow and `\(lh` for a diagonal left-up (home) arrow.

`cw` recognizes five request lines, as well as user-defined delimiters. The request lines look like `otroff`(1) macro requests, and are copied in their entirety by `cw` onto its output; thus, they can be defined "by the user" as `otroff`(1) macros; in fact, the `.CW` and `.CN` macros *should* be so defined (see HINTS below). The five requests are:

`.CW`
: Start of text to be set in the CW font; `.CW` causes a break; it can take precisely the same options, in precisely the same format, as are available on the `cw` command line.

`.CN`
: End of text to be set in the CW font; `.CN` causes a break; it can take the same options as are available on the `cw` command line.

`.CD`
: Change delimiters and/or settings of other options; takes the same options as are available on the `cw` command line.

.CP *arg1 arg2 arg3 ...* argn
All the arguments that are delimited like otroff(1) macro arguments are concatenated, with the odd-numbered arguments set in the CW font and the even-numbered ones in the prevailing font.

.PC *arg1 arg2 arg3 ...* argn
Same as .CP, except that the even-numbered arguments are set in the CW font and the odd-numbered ones in the prevailing font.

The .CW and .CN requests are meant to bracket text (e.g., a program fragment) that is to be typeset in the CW font as is. Normally, cw operates in the "transparent" mode. In that mode, except for the .CD request and the nine special four-character names listed in the table preceding, every character between .CW and .CN request lines stands for itself. In particular, cw arranges for periods (.) and apostrophes (') at the beginning of lines, and backslashes (\) everywhere to be hidden from otroff(1). The transparent mode can be turned off (see below), in which case normal otroff(1) rules apply; in particular, lines that begin with . and ' are passed through untouched (except if they contain delimiters). In either case, cw hides the effect of the font changes generated by the .CW and .CN requests; cw also defeats all ligatures (fi, ff, etc.) in the CW font.

The only purpose of the .CD request is to allow the changing of various options other than just at the beginning of a document.

The user can also define *delimiters*. The left and right delimiters perform the same function as the .CW/.CN requests; they are meant, however, to enclose CW words or phrases in running text (see example under LIMITATIONS below). cw treats text between delimiters in the same manner as text enclosed by .CW/.CN pairs, except that, for aesthetic reasons, spaces and backspaces inside .CW/.CN pairs have the same width as other CW characters, while spaces and backspaces between delimiters are half as wide, so they have the same width as spaces in the prevailing text (but are *not* adjustable). Font changes due to delimiters are *not* hidden.

Delimiters have no special meaning inside .CW/.CN pairs.

Typical usage for the cw command is:

```
cw files | otroff ...
```

checkcw checks that left and right delimiters, as well as the .CW/.CN pairs, are properly balanced. It prints out all offending lines.

Typical definitions of the .CW and .CN macros meant to be used with the mm(1) macro package:

```
.de CW
.DS I
.ps 9
```

```
.vs 10.5p
.ta 16m/3u 32m/3u 48m/3u 64m/3u 80m/3u 96m/3u ...
..
.de CN
.ta 0.5i 1i 1.5i 2i 2.5i 3i 3.5i 4i 4.5i 5i 5.5i 6i
.vs
.ps
.DE
..
```

At the very least, the `.CW` macro should invoke the `otroff`(1) no-fill (`.nf`) mode.

When set in running text, the CW font is meant to be set in the same point size as the rest of the text. In displayed matter, on the other hand, it can often be profitably set one point smaller than the prevailing point size (the displayed definitions of `.CW` and `.CN` above are one point smaller than the running text on this page). The CW font is sized so that when it is set in 9-point there are 12 characters per inch.

Documents that contain CW text may also contain tables and/or equations. If this is the case, the order of preprocessing should be: `cw`, `tbl`, and `eqn`. Usually, the tables contained in such documents will not contain any CW text, although it is entirely possible to have elements of the table set in the CW font; of course, care must be taken that `tbl`(1) format information not be modified by `cw`. Attempts to set equations in the CW font are not likely to be either pleasing or successful.

In the CW font, overstriking is most easily accomplished with backspaces: letting ← represent a backspace, d←←† yields d†. Because spaces (and, therefore backspaces) are half as wide between delimiters as inside `.CW`/`.CN` pairs (see above), two backspaces are required for each overstrike between delimiters.

EXAMPLES

The command:

```
cw text | tbl | otroff -mm
```

processes the text file `text`, sends the output to `tbl`(1), and then sends the output for final formatting to `otroff`(1) and `mm`(1).

WARNINGS

If text preprocessed by `cw` is to make any sense, it must be set on a typesetter equipped with the CW font or on a STARE facility; on the latter, the CW font appears as bold, but with the proper CW spacing.

Do not use periods (.), backslashes (\), or double quotes (") as delimiters, or as arguments to `.CP` and `.PC`.

Do not use cw with nroff, since nroff already makes everything constant-width.

LIMITATIONS

Certain CW characters don't concatenate gracefully with certain Roman characters, for example, a CW ampersand (&) followed by a Roman comma (,). In such cases, judicious use of otroff(1) half- and quarter-spaces (\| and \^) is most salutary; for example, one should use _&_\^ (rather than just plain _&_) to obtain & (assuming that _ is used for both delimiters).

The output of cw is hard to read. See also LIMITATIONS under otroff(1).

FILES

/bin/cw
: Executable file

/usr/lib/font/ftCW
: Font file

SEE ALSO

eqn(1), mmt(1), tbl(1), troff(1)

mm(5), mv(5) in *A/UX Programmer's Reference*

"Other Text Processing Tools" in *A/UX Text Processing Tools*

NAME

cxref — generates a C program cross-reference

SYNOPSIS

cxref [-c] [-o *file*] [-s] [-t] [-w[*num*]] *file...*

ARGUMENTS

-c Prints a combined cross-reference of all input files.

file Specifies the file from which a cross-reference is to be generated.

-o *file*
Directs output to named *file*.

-s Operates silently; does not print input filenames.

-t Formats listing for 80-column width.

-w[*num*]
Specifies the width option that formats output no wider than *num* (decimal) columns. This option will default to 80 if *num* is not specified or is less than 51.

DESCRIPTION

cxref analyzes a collection of C files and attempts to build a cross-reference table. The cxref command utilizes a special version of cpp to include information from #define statements in its symbol table. It produces a separate listing on standard output of all symbols (auto, static, and global) in each file, or with the -c option, of all symbols in combination. Each symbol contains an asterisk (*) before the declaring reference.

The -D, -I, and -U options are identical to the corresponding options in cc(1).

LIMITATIONS

The cxref command considers a formal argument in a #define macro definition to be a declaration of that symbol. For example, a program that contains the line

```
#include <ctype.h>
```

will contain many declarations of the variable c.

When using the -o option, the space between the -o and the *file* argument is critical. If you omit the space, as in

```
cxref -otest test.c
```

then the input file test.c will be destroyed.

STATUS MESSAGES AND VALUES

Error messages are unusually cryptic, but usually mean that you can't compile these files anyway.

FILES

`/usr/bin/cxref`
: Executable file

`/usr/lib/xpass`
: File that parses the input file

`/usr/lib/xcpp`
: File containing a special version of C-preprocessor

SEE ALSO

cc(1)

NAME

daps — invokes the Autologic APS-5 phototypesetter troff post-processor

SYNOPSIS

daps [-b] [-h*string*] [-o*list*] [-r] [-s*n*] [-t] [-w] [*file*]...

ARGUMENTS

-b Reports whether the phototypesetter is busy; does not print output.

file Specifies the file to be printed. If you do not specify a *file,* the standard input is printed. The specified *files* that you submit to daps should be prepared under the -Taps option of troff.

-h*string*
Prints *string* in this job's header. The header appears on a page preceding the output.

-o*list*
Prints pages whose numbers are given in the *list.* The list contains single numbers *n* and ranges *n1-n2.* A missing *n1* means the lowest numbered page, a missing *n2* means the highest.

-r Reports the number of 11-inch pages generated by this job. The daps command continues when you push the PROCEED button on the phototypesetter.

-s*n*
Stops after every *n* pages of output.

-t Directs output to the standard output instead of the phototypesetter.

-w Waits for phototypesetter to become free, then prints the output.

DESCRIPTION

daps prints *files* created by troff(1) on an Autologic APS-5 phototypesetter.

LIMITATIONS

Installations with an Autologic APS-5 phototypesetter should be aware that getting a good match to their Autologic fonts will almost certainly require hand-tuning of the distributed font description files (see the ''Files'' section).

FILES

/usr/bin/daps
Executable file

/dev/aps
APS-5 phototypesetter device

/usr/lib/font/devaps/*
Description files for APS-5

SEE ALSO

grap(1), mmt(1), mvt(1), pic(1), tc(1), troff(1)

NAME

`date` — displays and sets the date

SYNOPSIS

`date` [*mmddhhmm*[*yy*]] [+*format*]

ARGUMENTS

+*format*

Puts the output of `date` under the control of the user. Replace *format* with the date and time as described in the argument below.

mmddhhmm[*yy*]

Specifies the date and time. Replace the first *mm* with the month number, replace *dd* with the day number in the month, replace *hh* with the hour number (24-hour system), replace the second *mm* with the minute number, and replace *yy* with the last 2 digits of the year number (this is optional). If a number less than 70 is given, the year that results is 1970. For example,

```
date 10080045
```

sets the date to Oct. 8, 12:45 AM. The current year is the default if no year is mentioned.

DESCRIPTION

`date` displays the current date and time if no argument is given, or if the argument begins with +. Otherwise, the current date is set. The operating system operates in GMT. The `date` command takes care of the conversion to and from local standard and daylight time.

All output fields are of fixed size (zero padded if necessary). Each field descriptor is preceded by `%` and will be replaced in the output by its corresponding value. A single `%` is encoded by `%%`. All other characters are copied to the output without change. The string is always terminated with a newline character.

The Field Descriptors follow:

`n` Inserts a newline character

`t` Inserts a tab character

`m` Month of year—01 to 12

`d` Day of month—01 to 31

`y` Last 2 digits of year—70 to 99

`D` Date as mm/dd/yy

`H` Hour—00 to 23

`M` Minute—00 to 59

`S` Second—00 to 59

`T` Time as HH:MM:SS

`j` Day of year—001 to 366

`w` Day of week—Sunday = 0

`a` Abbreviated weekday—Sun to Sat

`h` Abbreviated month—Jan to Dec

`r` Time in AM/PM notation

EXAMPLES

The command:

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

generates the output:

```
DATE: 08/01/76
TIME: 14:45:05
```

WARNINGS

It is bad practice to change the date while the system is running multiuser.

STATUS MESSAGES AND VALUES

`No permission`
: You are not the superuser and you tried to change the date.

`bad conversion`
: The date that was set is syntactically incorrect.

`bad format character`
: The field descriptor is not recognizable.

FILES

`/bin/date`
: Executable file

`/etc/wtmp`
: File that records time

SEE ALSO

`gettimeofday`(2), `stime`(2), `time`(2), `printf`(3S), `utmp`(4) in *A/UX Programmer's Reference*

NAME

dbx — debugs and executes programs

SYNOPSIS

dbx [-c *file*] [-D] [-i] [-I *dir*]... [-r] [*objfile* [*coredump*]]

ARGUMENTS

-c *file*
: Executes the dbx commands in the specified file before reading from standard input.

-D
: Installs Macintosh low-memory globals into the dbx symbol table, allowing dbx to disassemble A-line traps generated for A/UX Toolbox routines. This option is useful for debugging applications that call the A/UX Toolbox.

-i
: Causes dbx to act as though standard input is a terminal. This option enables you to manually input dbx commands from the keyboard to the command line.

-I *dir*
: Adds the specified directory, *dir*, to the list of directories that are searched when dbx is looking for a source file. Normally dbx looks for source files in the current directory and in the directory where *objfile* is located. The directory search path can also be set with the use command.

objfile [*coredump*]
: Specifies the object file to debug. Such files are produced by a compiler with the appropriate flag (usually -g), which places symbol information in the object file.

  The object file contains a symbol table that includes the names of all the source files translated by the compiler to create it. These files are available for perusal while you are using the debugger.

  If a file named core exists in the current directory or if a *coredump* file is specified, dbx can be used to examine the state of the program when it was terminated.

-r
: Causes dbx to execute objfile before accepting debugging commands. If objfile runs successfully as indicated by a zero exit status, dbx exits. Otherwise, dbx reports the reason for termination and offers you the option of either entering the debugger or letting the program terminate. When this option is specified and the standard input is not a terminal, dbx reads from /dev/tty.

  Unless this option is specified, dbx waits for an explicit run command or another debugger command.

DESCRIPTION

dbx is a tool for source-level debugging and execution of programs under UNIX®. The machine-level facilities of dbx can be used on any program.

If the file dbxinit exists in the current directory, the debugger commands in it are executed. Your home directory will also be checked for a dbxinit file if one does not exist in the current directory.

When you are debugging A/UX applications that use the A/UX Toolbox, it is useful to use the -D option and to ignore the IOT, IO, and URG signals. The ignore command is described in the next section, ''Execution and Tracing Commands.''

Execution and Tracing Commands

The execution and tracing commands follow.

call *procedure*(*parameters*)
: Executes the object code associated with the named procedure or function.

catch *number*
catch *signal-name*
ignore *number*
ignore *signal-name*
: Start or stop trapping of the signal before it is sent to the program. This method is useful when a program being debugged handles interrupt signals. You can specify a signal by number or by a name (for example, SIGINT). Signal names are case insensitive and the SIG prefix is optional. By default all signals are trapped except SIGCONT, SIGCHILD, SIGALRM, and SIGKILL. If you are debugging A/UX applications that call the A/UX Toolbox, it is advisable to ignore SIGIOT, SIGIO, and SIGURG because these signals are used by A/UX Toolbox routines.

cont *integer*
cont *signal-name*
: Continue execution from the point at which it stopped. If a signal is specified, the process continues as though it received the signal. Otherwise, the process continues as though it had not been stopped.

  Execution cannot continue if the process called the standard procedure exit. The dbx program does not allow the process to exit, so you can examine the program state.

delete *command-number...*
: Removes the trace or stop corresponding to each given *command-number* argument. (You can print the numbers associated with traces and stops can be printed by using the status command.)

`next`
Executes up to the next source line. The difference between `next` and `step` is that if the line contains a call to a procedure or function, the `step` command will stop at the beginning of that block, whereas the `next` command will not.

`return` [*procedure*]
Continues until a return to *procedure* is executed if *procedure* is specified. If no procedure is specified, execution continues until the current procedure returns.

`run` [*args*] [<*file*][>*file*]
`rerun` [*args*] [<*file*][>*file*]
Start executing *objfile*, passing the *args* arguments as command-line arguments. The symbols < and > can be used to redirect input or output in the usual manner. When `rerun` is used without any arguments, the previous argument list is passed to the program; otherwise this command is identical to `run`. If *objfile* argument has been written since the last time the symbolic information was read, `dbx` will read the new information.

`status >` [*filename*]
Prints the currently active `trace` and `stop` commands.

`step`
Executes one source line.

`stop if` *condition*
`stop at` *source-line-number* [`if` *condition*]
`stop in` *procedure-or-function* [`if` *condition*]
`stop` *variable* [`if` *condition*]
Stop execution when the given line is reached, the given procedure or function is called, the given variable is changed, or the given condition is true.

`trace` [`in` *procedure-or-function*] [`if` *condition*]
`trace` *source-line-number* [`if` *condition*]
`trace` *procedure-or-function* [`in` *procedure-or-function*] [`if` *condition*]
`trace` *expression* `at` *source-line-number* [`if` *condition*]
`trace` *variable* `in` *procedure-or-function* [`if` *condition*]
Print tracing information when the program is executed. A number is associated with the command that is used to turn the tracing off. (See the description of `delete` command later in this list.)

The first argument describes what is to be traced. If it is a *source-line-number* argument, then the line is printed immediately before it is executed. Numbers for source line in a file other than the

current one must be preceded by the name of the file in quotation marks and a colon, as shown here:

```
"mumble.p":17
```

If the argument is a procedure or function name, every time the procedure or function is called, dbx displays information that tells you what routine called it, from what source line it was called, and what parameters were passed to it. In addition, its return is noted and, if it is a function, the value it returns is also printed.

If the argument is an expression with an `at` clause, the value of the expression is printed when the identified source line is reached.

If the argument is a variable, the name and value of the variable are printed whenever they change. Execution is substantially slower during this form of tracing.

If no argument is specified, all source lines are printed before they are executed. Execution is substantially slower during this form of tracing.

The clause `in` *procedure-or-function* causes tracing information to be printed only while dbx is being executed inside the given procedure or function.

The *condition* argument is a Boolean expression evaluated before the tracing information is printed. If the value is `FALSE`, the information is not printed.

Variables and Expressions

The dbx debugger resolves variables first in terms of the static scope of the current function, then in terms of the dynamic scope if the name is not defined in the static scope. If static and dynamic searches do not yield a result, an arbitrary symbol is chosen and the message ''[using *qualified name*]'' is printed. You can override the name-resolution procedure by qualifying an identifier with a block name, for example, *module*.*variable*. For C, source files are treated as modules named by the filename without the `.c` extension.

Expressions are specified with an approximately common subset of C and Pascal syntax. You can denote indirection by using either a prefix asterisk (`*`) or a postfix caret (`^`). Array expressions are subscripted by brackets (`[ ]`). The field reference dot operator (`.`) can be used with pointers as well as records, making the C arrow operator (`->`) unnecessary (although it is supported).

Types of expressions are checked. You can override the type of an expression by using the *type-name*(*expression*) construct. When there is no corresponding named type, you can use the special constructs `&`*type-name* and `$$`*tag-name* to represent a pointer to a named type or C structure tag.

The commands are:

`assign` *variable=expression*
: Assigns the value of the expression to the variable.

`dump` [*procedure*] `>` [*filename*]
: Prints the names and values of variables in the given procedure, or the current one if none is specified. If the replacement value for *procedure* is supplied as period (.), then the names and values of all active variables are printed.

`print` *expression*[`,` *expression*]...
: Prints the values of the given expressions.

`up` [*count*]
`down` [*count*]
: Move the current function, which is used for resolving names, up or down the stack *count* levels. The default for *count* is 1.

`whatis` *name*
: Prints the declaration of the given name, which can be qualified with block names as described earlier.

`where`
: Prints a list of the active procedures and functions.

`whereis` *identifier*
: Prints the full qualification of all symbols whose names match the given identifier. The order in which the symbols are printed is not meaningful.

`which` *identifier*
: Prints the full qualification of the given identifier, that is, the outer blocks with which the identifier is associated.

Accessing Source Files vs Navigating through

You can access source files by using these commands instead of navigating through your source code:

`/`*regular expression*[`/`]
`?`*regular expression*[`?`]
: Search forward or backward in the current source file for the given pattern.

`edit` [*filename*]

`edit` *procedure-or-function-name*
Invoke an editor on the specified file or on the current source file if no *filename* argument is specified. If a procedure or function name is specified, the editor is invoked on the file that contains that procedure or function. Which editor is invoked by default depends on the installation. You can override the default by setting the environment variable `EDITOR` to the name of the desired editor.

`file` [*filename*]
Changes the current source filename to the specified filename. If none is specified, the current source filename is printed.

`func` [*procedure-or-function*]
Changes the current function to the one specified. If none is specified, this command prints the current function. Changing the current function implicitly changes the current source file to the one that contains the function; it also changes the current scope used for name resolution.

`list` [*source-line-number1*[`,` *source-line-number2*]]
`list` *procedure-or-function*
List the lines in the current source file from the specified *source-line-number1* to *source-line-number2*, inclusive. If no lines are specified, the next `$listwindow` lines (the default is 10) are listed. If the name of a procedure or function is given, lines *n-k* to *n+k* are listed, where *n* is the first statement in the procedure or function and *k* is defined by the variable `$listwindow`.

`use` *directory-list*
Sets the list of directories to be searched when looking for source files. (Also see the description of the `-I` option in the ''Arguments'' section earlier in this manual page.) If no *directory-list* argument is provided for the `use` command, the current directory list is displayed.

Command Aliases and Variables
The command aliases and variables commands follow.

`alias` *name name*
`alias` *name* `"`*string*`"`
`alias` *name*`(`*parameters*`)` `"`*string*`"`
Check to see if *name* is an alias for either a command or a string when commands are processed. If it is an alias, then treats the input as though the corresponding string (with values substituted for any parameters) had been entered. For example, the command

```
alias rr rerun
```

can be used to define an alias `rr` for the `rerun` command.

The command

```
alias b(x)
```

can be used to define an alias `b` that sets a stop at the passed line number (*x*). For example, the command

```
b(12)
```

expands to

```
stop at 12
```

when `b` is aliased, as shown earlier.

`set` *name*[=*expression*]
Defines values for debugger variables. The names of these variables must not conflict with names in the program being debugged, and are expanded to the corresponding expression within other commands. The following variables have a special meaning:

`$frame`
Causes `dbx` to use the stack frame pointed to by the address for performing stack traces and accessing local variables when this variable is set to an address. This facility is of particular use for kernel debugging.

`$hexchars`
`$hexints`
`$hexoffsets`
`$hexstrings`
Cause `dbx` to print out characters, integers, offsets from registers, or character pointers, respectively, in hexadecimal when one of these is set.

`$listwindow`
Specifies a number of lines to be listed by the `list` command. The default value is 10.

`$mapaddrs`
Causes `dbx` to start mapping addresses. Unsetting this variable causes address mapping to stop. As with `$frame`, this variable is useful for kernel debugging.

`$unsafecall`
`$unsafeassign`
Turn off strict type-checking for arguments to the subroutine or function calls (for example, in the `call` statement) when the `$unsafecall` variable is set. When the `$unsafeassign` variable is set, strict type-checking between the two sides of an `assign` statement is turned

off. These variables should be used with great care, because they severely limit the usefulness of dbx in detecting errors.

`unalias` *name*
Removes the alias with the given name.

`unset` *name*
Deletes the debugger variable associated with the specified *name*.

Machine-Level Commands

You specify symbolic addresses by preceding the name of a register with an ampersand (&). Registers are denoted by `$r`*n* where *n* is the number of the register. Addresses can be expressions made up of other addresses and the operators +, -, and indirection (unary *).

The machine-level commands as described below:

address,*address*/ [*mode*]
address/[*count*] [*mode*]
Print the contents of memory, starting at the first address and continuing up to the second address or until *count* items are printed. If the address is dot (.), the address following the one printed most recently is used. The *mode* argument specifies how memory is to be printed; if it is omitted, the mode last specified is used. The initial mode is X, which prints long words in hexadecimal. The following modes are supported:

- b Prints a byte in octal.
- c Prints a byte as a character.
- d Prints a short word in decimal.
- D Prints a long word in decimal.
- f Prints a single-precision real number.
- g Prints a double-precision real number.
- i Prints the machine instruction.
- o Prints a short word in octal.
- O Prints a long word in octal.
- s Prints a string of characters terminated by a null byte.
- x Prints a short word in hexadecimal.
- X Prints a long word in hexadecimal.

`stepi`
`nexti`
Step by a single instruction rather than by a source line. These variables single-step as the `step` and `next` variables do.

`tracei` [*address*] [`if` *cond*]
`tracei` [*variable*] [`at` *address*] [`if` *cond*]
`stopi` [*address*] [`if` *cond*]
`stopi` [`at`] [*address*] [`if` *cond*]
Turn on tracing or set a stop, using the machine-instruction address specified by *address*.

Miscellaneous Commands

`help`
Prints out a synopsis of `dbx` commands.

`quit`
Exits `dbx`.

`sh` *command-line*
Passes the command line to the shell for execution. The `SHELL` environment variable determines which shell is used.

`source` *filename*
Reads `dbx` commands from the specified file.

FILES

`.dbxinit`
File containing initial commands
`/usr/ucb/dbx`
Executable file
`a.out`
Object file

SEE ALSO

`c89`(1), `cc`(1)

"`dbx` Reference" in *A/UX Development Tools*

NAME

dc — desk calculator

SYNOPSIS

dc [*file*]

ARGUMENTS

file Specifies the file from which the input is used.

DESCRIPTION

dc is an arbitrary precision arithmetic package. Ordinarily it operates on decimal integers, but one may specify an input base, output base, and a number of fractional digits to be maintained. The overall structure of dc is a stacking (reverse Polish) calculator.

The following constructions are recognized:

number
: Pushes the value of *number* on the stack. A number is an unbroken string of one or more digits in the range 0-9. It may be preceded by an underscore (_) to indicate a negative number. Numbers may contain decimal points.

+ - / * % ^
: Operates on the top two values on the stack. These are added (+), subtracted (-), multiplied (*), divided (/), remaindered (%), or exponentiated (^). The two entries are popped off the stack; the result is pushed on the stack in their place. Any fractional part of an exponent is ignored.

s*x* Pops the top of the stack and stores it in a register named *x*, where *x* may be any character.

S*x* Pushes the value on *x*, which is treated as a stack.

l*x* Pushes the value in register *x* on the stack. The register *x* is not altered. All registers start with zero value.

L*x* Pops the top value of register *x*, which is treated as a stack, onto the main stack.

d Duplicates the top value on the stack.

p Prints the top value on the stack. The top value remains unchanged.

P Interprets the top of the stack as an ASCII string, removes it, and prints it.

f Prints all values on the stack.

q Exits the program. If executing a string, the recursion level is popped by two. Alternately, CONTROL-d (EOF) will exit from dc.

`Q` Pops the top value on the stack and pops the string execution level by that value. Alternately, CONTROL-d (EOF) will exit from `dc`.

`x` Treats the top element of the stack as a character string and executes it as a string of `dc` commands.

`X` Replaces the number on the top of the stack with its scale factor.

[*string*]
Puts the bracketed ASCII string onto the top of the stack.

<*x* >*x* =*x*
Pops the top two elements of the stack and compares them. Register *x* is evaluated if they obey the stated relation.

`v` Replaces the top element on the stack by its square root. Any existing fractional part of the argument is taken into account, but otherwise the scale factor is ignored.

`!` Interprets the rest of the line as a system command.

`c` Pops all values on the stack.

`i` Pops the top value on the stack and uses it as the number radix for further input.

`I` Pushes the input base on the top of the stack.

`o` Pops the top value on the stack and uses it as the number radix for further output.

`O` Pushes the output base on the top of the stack.

`k` Pops the top of the stack and uses that value as a non-negative scale factor: prints the appropriate number of places on output, and maintains them during multiplication, division, and exponentiation. The interaction of scale factor, input base, and output base will be reasonable if all are changed together.

`z` Pushes the stack level onto the stack.

`Z` Replaces the number on the top of the stack with its length.

`?` Takes a line of input from the input source (usually the terminal) and executes it.

`; :`
Allows `bc` to perform array operations.

EXAMPLES

The command:

```
dc
24.2 56.2 + p
```

adds the two numbers and prints the result (top value in the stack).

The command:

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

prints the first ten values of *n*!.

STATUS MESSAGES AND VALUES

`x is unimplemented`
: The argument, *x*, is an octal number.

`stack empty`
: Not enough elements on the stack to do what was asked.

`Out of space`
: The free list is exhausted (too many digits).

`Out of headers`
: Too many numbers being kept around.

`Out of pushdown`
: Too many items on the stack.

`Nesting Depth`
: Too many levels of nested execution.

FILES

`/usr/bin/dc`
: Executable file

SEE ALSO

`bc`(1)

"`dc` Reference," in *A/UX Programming Languages and Tools, Volume 2*

NAME

dd — converts and copies a file

SYNOPSIS

dd [bs=*n*] [cbs=*n*] [conv=ascii] [conv=ebcdic] [conv=ibm] [conv=lcase] [conv=noerror] [conv=swab] [conv=sync] [conv=*type*,*type*] [conv=ucase] [count=*n*] [ibs=*n*] [if=*file*] [multi=in] [multi=in,out] [multi=out] [of=*file*] [obs=*n*] [seek=*n*] [skip=*n*]

ARGUMENTS

bs=*n*
: Sets both input and output block size, superseding ibs and obs; also, if no conversion is specified, it is particularly efficient since no incore copy needs to be generated

cbs=*n*
: Specifies the conversion buffer size. Replaces *n* with the size of the buffer.

conv=ascii
: Converts EBCDIC text to ASCII text.

conv=ebcdic
: Converts ASCII text to EBCDIC text.

conv=ibm
: Specifies a slightly different map of ASCII to EBCDIC.

conv=lcase
: Maps alphabetics to lowercase.

conv=noerror
: Does not stop processing when an error occurs.

conv=swab
: Swaps every pair of bytes.

conv=sync
: Pads every input block to ibs.

conv=*type*,*type*
: Specifies several comma-separated conversions, where *type* is one of the conversions listed for conv.

conv=ucase
: Maps alphabetics to uppercase.

count=*n*
: Copies only *n* input blocks.

`ibs=`*n*
Inputs the block size, *n* bytes (the default 512)

`if=`*file*
Inputs the *file*. The standard input is default.

`multi=in`
Indicates that the input file is multivolume.

`multi=in,out`
Indicates that both the input file and the output file are multivolume.

`multi=out`
Indicates that the output file is multivolume.

`of=`*file*
Outputs the *file*. The standard output is default.

`obs=`*n*
Outputs the block size *n*. The default 512.

`seek=`*n*
Seeks *n* blocks from beginning of output file before copying. The `dd` command creates the specified output file (see `creat`(2)), which insures that the length of the file will be zero for regular files; seeking *n* blocks from the beginning of the output file will fill the skipped area with zeros (nulls)

`skip=`*n*
Skips *n* input blocks before starting to copy.

DESCRIPTION

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

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 separated by `x` to indicate a product.

The `cbs` option is used only if `ascii`, `ebcdic`, or `ibm` conversion is specified. In the first case, `cbs` characters are placed into the conversion buffer, converted to ASCII, and trailing blanks are trimmed and a newline added before sending the line to the output. In the next two cases, ASCII characters are read into the conversion buffer, converted to EBCDIC (or the IBM version of EBCDIC), and blanks are added to make up an output block of size `cbs`.

If multivolume input (output) is specified, a prompt is given at end-of-file to allow another volume to be mounted.

After completion, `dd` reports the number of whole and partial input and output blocks.

EXAMPLES

The command:

```
dd if=/dev/rmt/0m of=x ibs=800 cbs=80 conv=ascii,lcase
```

will read an EBCDIC tape blocked at ten 80-byte EBCDIC card images per block into the ASCII file `x`.

Note the use of raw magnetic tape. `dd` is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary block sizes.

LIMITATIONS

The ASCII/EBCDIC conversion tables are taken from the 256-character standard in the *CACM*, November, 1968. The `ibm` conversion, while less blessed as a standard, corresponds better to certain IBM print-train conventions. There is no universal solution.

Newlines are inserted only on conversion to ASCII; padding is done only on conversion to EBCDIC. These should be separate options.

When using `dd` to transfer data over an Ethernet connection, you should specify a block size of 1 kilobyte.

STATUS MESSAGES AND VALUES

The output:

f+*p* `blocks in(out)`

contains numbers of full and partial blocks read (written).

FILES

`/bin/dd`
Executable file

SEE ALSO

`cp`(1), `cpio`(1), `tar`(1), `tr`(1)

NAME

delta — makes a delta (change) to an SCCS file

SYNOPSIS

delta [-g*list*] [-m[*mrlist*]] [-n] [-p] [-r*SID*] [-s] [-y[*comment*]] *file...*

ARGUMENTS

file Specifies the file to be changed.

-g*list*

Specifies a *list* (see get(1) for the definition of *list*) of deltas which are to be *ignored* when the file is accessed at the change level (*SID*) created by this delta.

-m[*mrlist*]

Requires that a Modification Request (MR) number be supplied as the reason for creating the new delta, if the SCCS file has the v option set (see admin(1)).

If the -m option is not used and the standard input is a terminal, the prompt MRs? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. The MRs? prompt always precedes the comments? prompt (see -y option).

MRs in a list are separated by blanks and/or tab characters. An unescaped newline character terminates the MR list.

Note that if the v option has a value (see admin(1)), it is taken to be the name of a program (or shell procedure) which will validate the correctness of the MR numbers. If a nonzero exit status is returned from MR number validation program, delta terminates (it is assumed that the MR numbers were not all valid).

-n Specifies retention of the edited g-file (normally removed at completion of delta processing).

-p Causes delta to print (on the standard output) the SCCS file differences before and after the delta is applied in a diff(1) format.

-r*SID*
Identifies (uniquely) which delta is to be made to the SCCS file. The use of this keyletter is necessary only if two or more outstanding gets for editing (get -e) on the same SCCS file were done by the same person (login name). The *SID* value specified with the -r keyletter can be either the *SID* specified on the get command line or the *SID* to be made as reported by the get command (see get(1)). A message will display if the specified *SID* is ambiguous or omitted on the command line.

-s Suppresses the issue on the standard output of the created delta's *SID*, as well as the number of lines inserted, deleted, and unchanged in the SCCS file.

-y[*comment*]
Specifies arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*. If the comment includes spaces, you must enclose the entire string in double quotes.

-y is not specified and the standard input is a terminal, the prompt comments? is issued on the standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

DESCRIPTION

delta is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by get(1) (called the g-file, or generated file).

The delta command makes a delta to each named SCCS file. If a directory is named, delta behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored. If a hypen (-) is given, the standard input is read (see WARNINGS); each line of the standard input is taken to be the name of an SCCS file to be processed.

The delta command may issue prompts on the standard output depending upon certain keyletters specified and flags (see admin(1)) that may be present in the SCCS file (see -m and -y options above).

EXAMPLES

The command:

```
% delta s.test1.c
  comments?  second version
  1.2
  1 inserted
  0 deleted
  12 unchanged
```

does a `delta` on file `test1.c`.

STATUS MESSAGES AND VALUES

Use `help` for explanations.

WARNINGS

Lines beginning with an SOH ASCII character (binary 001) cannot be placed in the SCCS file unless the SOH is escaped. This character has special meaning to SCCS (see `sccsfile`(5)) and will cause an error.

A `get` of many SCCS files, followed by a `delta` of those files, should be avoided when the `get` generates a large amount of data. Instead, multiple `get`/`delta` sequences should be used.

If the standard input (`-`) is specified on the `delta` command line, the `-m` (if necessary) and `-y` options *must* also be present. Omission of these keyletters causes an error to occur.

Comments are limited to text strings of at most 512 characters.

FILES

`/usr/bin/delta`
: Executable file

g-file
: File that existed before the execution of `delta`; removed after completion of `delta`

p-file
: File that existed before the execution of `delta`; may exist after completion of `delta`

q-file
: File that was created during the execution of `delta`; removed after completion of `delta`

x-file
: File that was created during the execution of `delta`; renamed to SCCS file after completion of `delta`

z-file
File that was created during the execution of `delta`; removed during the execution of `delta`

d-file
File that was created during the execution of `delta`; removed after completion of `delta`

`/usr/bin/bdiff`
File containing the program to compute differences between the ''gotten'' file and the *g-file*

SEE ALSO

`admin`(1), `bdiff`(1), `cdc`(1), `get`(1), `help`(1), `prs`(1), `rmdel`(1), `sccs`(1)

`sccsfile`(4) in *A/UX Programmer's Reference*

''SCCS Reference,'' in *A/UX Programming Languages and Tools, Volume 2*

NAME

derez — decompiles a resource file

SYNOPSIS

derez [-c] [-d*macro-assignment*]... [-e] [-i*include-dir*]... [-m*string-size*] [-p] [-rd] [-u*macro*]... *resource-file* [*resource-description-file*]...

derez [-c] -o*scope* [-d*macro-assignment*]... [-e] [-i*include-dir*]... [-m*string-size*] [-p] [-rd] [-u*macro*]... *resource-file* [*resource-description-file*]...

derez [-c] -s*omit-scope* [-d*macro-assignment*]... [-e] [-i*include-dir*]... [-m*string-size*] [-p] [-rd] [-u*macro*]... *resource-file* [*resource-description-file*]...

ARGUMENTS

-c[ompatible]
: Generates output that is backward-compatible with rez 1.0.

-d[efine]*macro-assignment*
: Declares preprocessor macros in the form *name=value* that are equivalent to symbolic constants defined with

```
#define macro [value]
```

If no value is assigned, but a symbolic name is given, its value is set to the empty string (which still qualifies it as a ''defined'' macro).

-e[scape]
: Does not honor escape sequences that are normally honored (such as \0xff); prints these characters as extended Macintosh characters instead. Not all characters are defined in all fonts.

Normally, characters with values between 0x20 and 0xD8 are printed as Macintosh characters. When you specify the -e option, however, all characters (except null, newline, tab, backspace, form feed, vertical tab, and rubout) are printed as characters, not as escape sequences.

-i*include-dir*
: Searches the specified directories for any #include instructions inside the resource description file or files. The paths are searched in the order of their appearance on the command line.

To decompile an A/UX Toolbox resource file, use this command:

```
-i /:mac:lib:rincludes
```

`-m[axstringsize]`*string-size*
: Sets the maximum string size to *string-size*, which must be in the range 2-120. Use this option to set the maximum width of strings in the output.

`-o[nly]`*scope*
: Decompiles resources that fall within the scope specified only. See the description of *scope* later in this list. This option cannot be used if the `-s` option is used.

`-p` Displays progress and version information.

`-rd`
: Suppresses warning messages if a resource type is redeclared.

resource-description-file
: Specifies files containing the (source code) type declarations used to produce the resource file.

resource-file
: Specifies the name of the file containing the compiled resource. You must specify a resource file; `derez` does not read the standard input.

scope
omit-scope
: Declare the scope of resources to be either decompiled or skipped depending on whether the `-o` or the `-s` option is specified. (See the description of `-o` and `-s` earlier in this list.) The scope can be specified in several ways:

  resource-type
  : Limits the scope to the specified resource type, such as 'CODE'.

  " ' *resource-type* ' (*id1*) "
  : Limits the scope to a resource type and resource ID number.

  " ' *resource-type* ' (*id1* : *id2*) "
  : Limits the scope to a resource type and a range of resource ID numbers.

  " ' *resource-type* ' *resource-name* "
  : Limits the scope to a particular resource name of a particular resource type.

`-s[kip]`*scope*
: Skips the decompilation of resources that fall within the scope specified. See the descriptions of *scope* and *omit-scope* later in this list. This option cannot be used if the `-o` option is used.

  For example, you can save execution time by skipping CODE resources. You can repeat the `-s` option any number of times.

`-u[ndef]`*macro*
Undefines the macro variable *macro*. You can do the same thing by writing this at the beginning of the input file:

```
#undef macro
```

DESCRIPTION

`derez` creates a text representation (resource description) of a compiled resource file according to the resource type declarations in the resource description files. The resource description is written to standard output. If the output of `derez` is used as input to `rez` along with the same resource description files, it produces the same resource file that was originally input to `derez`.

To help generate resource code, certain basic data-type information is read from the files you specify as the resource description file or files. The type declarations in these resource description files should consist of `resource` and `data` statements just like those that can be understood by the `rez` program. The `derez` command ignores all `include` (but not `#include`), `read`, `data`, and `resource` statements found in the resource description file. (It still parses these statements for correct syntax.) Appendix C, ''Resource Compiler and Decompiler,'' in *A/UX Toolbox: Macintosh ROM Interface,* describes the format of resource type declarations.

The type declarations for standard Macintosh resources are contained in the files `types.r` and `systypes.r` in the directory `/mac/lib/rincludes`.

If you do not specify a resource description file, the output consists of data statements giving the resource data in hexadecimal form, without any additional format information.

The `derez` command is not guaranteed to be able to run a declaration backward. If `derez` does not run a declaration backward, the command produces a `data` statement instead of the appropriate `resource` statement.

EXAMPLES

Using the definitions in the file `/mac/lib/rincludes/types.r`, this command decompiles the resource file `%sample` and writes the output to the file `sample.r`:

```
derez -i /mac/lib/rincludes sample types.r > sample.r
```

You do not need to use quotation marks to specify a literal type as long as that type starts with a letter. Do not use escape characters or other special characters. This example specifies the type 'MENU':

```
derez -only MENU ...
```

If a resource ID, range of IDs, or resource name is given, the entire option parameter must appear inside single quotation marks, as in this example:

```
derez -only "'MENU'(1:128)"
```

If `derez` has access to the type definitions, it generates more meaningful output. For example, this command displays all of the 'MENU' resources in `%sample`, and the type definition for 'MENU' resources is in the file `types.r`:

```
derez -o MENU -i /mac/lib/rincludes sample types.r
```

STATUS MESSAGES AND VALUES

If no errors or warnings are detected, `derez` runs silently. Errors and warnings are written to standard error output. (See `intro`(3S) in *A/UX Programmer's Reference*.)

The `derez` command returns one of these status values:

0 No errors

1 Error in parameters

2 Syntax error in file

3 I/O or program error

LIMITATIONS

The `derez` command is not supported in 24-bit mode; you can use it only in 32-bit mode. You must run it from the command line while you are logged in to the Macintosh environment.

FILES

`/mac/bin/derez`
Executable file

SEE ALSO

`rez`(1)

`intro`(3s) in *A/UX Programmer's Reference*

A/UX Toolbox: Macintosh ROM Interface

NAME

deroff — removes nroff/troff, tbl, and eqn constructs

SYNOPSIS

deroff [-m*x*] [-w] [*file*]...

ARGUMENTS

file Specifies the file containing the constructs that are to be removed. If this argument is not given, deroff reads the standard input.

-m*x* Specifies the macro package. Replace *x* with an m, s, or l. If you select the -mm macro package, the macros will be interpreted so that only running text is output (that is, no text from macro lines). If the -ml macro package is chosen, the -mm macro package is forced, and also causes deletion of lists associated with the mm macros.

-w Causes the output to be a word list, one *word* per line, with all other characters deleted. Otherwise, the output follows the original, with the deletions mentioned above. In text, a *word* is any string that contains at least two letters and is composed of letters, digits, ampersands (&), and apostrophes ('); in a macro call, however, a *word* is a string that *begins* with at least two letters and contains a total of at least three letters. Delimiters are any characters other than letters, digits, apostrophes, and ampersands. Trailing apostrophes and ampersands are removed from words.

DESCRIPTION

deroff reads each of the *file*s in sequence, removes all troff(1) requests, macro calls, backslash constructs, eqn(1) constructs (between .EQ and .EN lines, and between delimiters), and tbl(1) descriptions, perhaps replacing them with white space (blanks and blank lines), and writes the remainder of the file on the standard output. The deroff command follows chains of included files (.so and .nx troff commands); if a file has already been included, a .so naming that file is ignored and a .nx naming that file terminates execution.

EXAMPLES

The command:

```
deroff textfile
```

removes all nroff, troff, and macro definitions from textfile.

LIMITATIONS

The deroff command is not a complete troff interpreter, so it can be confused by subtle constructs. Most such errors result in too much rather than too little output.

The -ml option does not handle nested lists correctly.

FILES

/usr/bin/deroff
Executable file

SEE ALSO

eqn(1), nroff(1), tbl(1), troff(1)

NAME

df — reports the used and unused storage capacity for a file system

SYNOPSIS

df -t [-f] [-T *fs-type*] [*fs-reference*]...

df -B [-i] [-T *fs-type*] [*fs-reference*]...

df -p [-i] [-T *fs-type*] [*fs-reference*]...

ARGUMENTS

-B Lists the free disk blocks in the BSD style of output format.

-f Computes the number of blocks in the free list (for SVFS file systems only).

fs-reference
Specifies the file system to query. The reference can be constructed in terms of a file or directory that resides on the file system you want to query or in terms of a device file that corresponds to the file system you want to query.

-i Lists statistics regarding inodes along with everything else reported. This option can only be used with the -p and -B options.

-p Lists the free disk blocks in the POSIX style of output format (POSIX User Portability Extension).

-t Reports the total allocated block and inode figures along with everything else. This option is not available when the -p or -B options are used.

-T *fs-type*
Establishes the type of file system as *file-system-type*. The accepted types are: 4.2 and 5.2. See fstab(4) for more detailed information regarding file system types.

DESCRIPTION

df reports the current state of a file system in terms of its data storage capacities. You may specify the file system reference (*fs-reference*) in two ways. File systems may be specified either as device files (for example, /dev/dsk/c0s0d0) or as filenames (for example, /usr). If *fs-reference* is given in the form of a regular file or directory, df reports on the amount of free space for the file system on which that file resides. Without any arguments, df displays the amount of free blocks and free inodes for all of the mounted file systems.

For file systems based upon the Berkeley Software Distribution (BSD) model, the remaining free space is calculated differently depending on whether you are using the root account or not. The root account is always allowed access to all the available disk blocks. Because BSD file systems

cannot be used effectively without a residual amount of free disk space, most users are denied access to a fixed percentage of the disk blocks. The amount by which free blocks are reduced for normal users is controlled through a `tunefs`(1M) parameter. However, you should not reduce this parameter to less than five percent because of the increased possibility of fragmentation resulting in impaired performance.

LIMITATIONS

Since inodes are file system dependent, the number of inodes reported on remotely mounted file systems is always zero.

FILES

`/bin/df`
: Executable file

`/dev/dsk/*`
: Disk partition device files

`/etc/mtab`
: File containing list of currently mounted file systems

SEE ALSO

`mount`(1M) in *A/UX System Administrator's Reference* `fs`(4), `fstab`(4), `mtab`(4) in *A/UX Programmer's Reference*

NAME

diction, explain — locate wordy sentences in a document

SYNOPSIS

diction [-f *pfile*] [-ml] [-mm] *file*...

diction [-ml] [-mm] [-n] *file*...

explain

ARGUMENTS

-f *pfile*
: Allows you to supply your own pattern file to be used in addition to the default *file*. Replace *pfile* with a list of (wordy) phrases, with one phrase per line. The default *pfile* is /usr/lib/dict.d.

file
: Specifies the file to be searched.

-ml
: Causes deroff to skip lists. The option should be used if the document contains many lists of nonsentences.

-mm
: Specifies the -mm macro package. This option overrides the -ms macro package, which is the default.

-n
: Suppresses the default *pfile*.

DESCRIPTION

diction finds all sentences in a document that contain phrases from a data base of bad or wordy diction. Each phrase is bracketed with []. Because diction runs deroff before looking at the text, formatting header files should be included as part of the input.

explain is an interactive thesaurus for the phrases found by diction. It prompts you with:

```
phrase?
```

to which you should respond by typing the *phrase* flagged by diction that you need explained. The explanation tells what to use instead of *phrase*. To get out of explain, press DELETE.

LIMITATIONS

Use of nonstandard formatting macros may cause incorrect sentence breaks. In particular, diction does not recognize the -me macro package.

FILES

/usr/ucb/diction
: Executable file

`/usr/ucb/explain`
Executable file

SEE ALSO
`deroff`(1), `style`(1), `spell`(1)

NAME

diff — compares two files or directories for any differences

SYNOPSIS

diff [-b] [-c] [-e] [-f] [-h] [-l] [-r] [-s] [-S*name*] *dir1 dir2*

diff [-b] [-c] [-e] [-f] [-h] *file1 file2*

diff [-b] *file1 file2*

ARGUMENTS

-b Causes trailing blanks (spaces and tabs) to be ignored, and other strings of blanks to compare equal.

-c Produces a diff with lines of context. The default is to preset 3 lines of context that may be changed, for example, to 10, by -c10. With the -c option, the output format is modified slightly: the output beginning with identification of the files involved and their creation dates and then each change is separated by a line with a dozen *'s. The lines removed from *file1* are marked with "–"; those added to *file2* are marked "+". Lines which are changed from one file to the other are marked in both files with "!".

dir1 Specifies the directory to be compared with *dir2*.

dir2 Specifies the directory to be compared with *dir1*.

-D*string*
Causes diff to create a merged version of *file1* and *file2* on the standard output, with C preprocessor controls included so that a compilation of the result without defining *string* is equivalent to compiling *file1,* while defining *string* will yield *file2.*

-e Produces a script of *a*, *c*, and *d* commands for the editor ed, which will recreate *file2* from *file1*. In connection with the -e option, the following shell program may help maintain multiple versions of a file. Only an ancestral file ($1) and a chain of version-to-version ed scripts ($2,$3, . . .) made by diff need be on hand. A "latest version" appears on the standard output.

```
(shift; cat $*; echo '1,$p') | ed - $1
```

Extra commands are added to the output when comparing directories with the -e, option so that the result is an sh(1) script for converting text files which are common to the two directories from their state in *dir1* to their state in *dir2*. Since such a shell script is useful only in a file that you may run on other files, it is best to redirect the output of this command into a file.

-f Produces a script similar to that of the -e option not useful with ed, and in the opposite order.

file1
Specifies the file to be compared with *file2*. If *file1* is not a directory, it can be given a "-", in which case uses the standard input. If *file1* is a directory, then a file in that directory whose filename is the same as the filename of *file2* is used.

file2
Specifies the file to be compared with *file1*. If *file2* is not a directory, it can be given a "-", in which case uses the standard input. If *file2* is a directory, then a file in that directory whose filename is the same as the filename of *file1* is used.

`-h` Does a fast, half-hearted job. It works only when changed stretches are short and well-separated, but does work on files of unlimited length.

`-l` Specifies long output format; each text file `diff` is piped through `pr` to paginate it, other differences are remembered and summarized after all text file differences are reported.

`-r` Causes application of `diff` recursively to common subdirectories encountered.

`-s` Causes `diff` to report files which are the same, which are otherwise not mentioned.

`-S`*name*
Starts a directory `diff` in the middle beginning with file *name.*

DESCRIPTION

If both arguments are directories, `diff` sorts the contents of the directories by name, and then runs the regular file `diff` algorithm (described below) on text files which are different. Binary files which differ, common subdirectories, and files which appear in only one directory are listed.

When run on regular files, and when comparing text files which differ during directory comparison, `diff` tells what lines must be changed in the files to bring them into agreement. Except in rare circumstances, `diff` finds a smallest sufficient set of file differences.

There are several options for output format; the default output format contains lines of these forms:

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

These lines resemble `ed` commands to convert *file1* into *file2*. The numbers after the letters pertain to *file2*. In fact, by exchanging `a` for `d` and reading backward, one may ascertain equally how to convert *file2* into *file1*.

As in ed, identical pairs where *n1* = *n2* or *n3* = *n4* are abbreviated as a single number.

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

LIMITATIONS

Editing scripts produced under the -e or -f option are naive about creating lines consisting of a single ".".

When comparing directories with the -b option specified, diff first compares the files as with cmp, and then decides to run the diff algorithm if they are not equal. This may cause a small amount of spurious output if the files then turn out to be identical, because the only differences are insignificant blank string differences.

If an unrecognized option is specified, diff performs the default operation anyway.

The diff command may not work if files contain a very long line, or if files are very long.

STATUS MESSAGES AND VALUES

Exit status is 0 for no differences, 1 for some, 2 for trouble.

FILES

/usr/bin/diff
: Executable file

/tmp/d?????
: Temporary file

/usr/lib/diffh
: Executable file

/bin/pr
: Executable file

SEE ALSO

bdiff(1), cmp(1), comm(1), cpp(1), diff3(1), ed(1)

"Other Tools" in *A/UX Programming Languages and Tools, Volume 2*

NAME

diff3 — compares three versions of a file

SYNOPSIS

diff3 [-3] [-e] [-x] *file1 file2 file3*

ARGUMENTS

-3 Produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

This option is equivalent to the -x option.

-e Causes diff3 to publish a script for the editor ed which results in all changes from *file2* and *file3* being implemented into file 1. that is, the changes that normally would be flagged ==== and ====3.

file1

Specifies the first version of the file.

file2

Specifies the second version of the file.

file3

Specifies the third version of the file.

-x Produces a script to incorporate only changes flagged ==== (====3). The following command will apply the resulting script to *file1*.

```
(cat script; echo '1,$p') | ed - file1
```

This option is equivalent to the -3 option.

DESCRIPTION

diff3 compares three versions of a file, and publishes disagreeing ranges of text flagged with these codes:

====

all three files are different

====1

file1 is different

====2

file2
is different

====3

file3 is different

The type of change suffered in converting a given range of a given file to some other is indicated in one of these ways:

f : *n1* a
Text is to be appended after line number *n1* in file *f*, where *f* = 1, 2, or 3.

f : *n1* , *n2* c
Text is to be changed in the range line *n1* to line *n2*. If *n1* = *n2*, the range may be abbreviated to *n1*.

The original contents of the range follows immediately after a c indication. When the contents of two files are identical, the contents of the lower-numbered file is suppressed.

EXAMPLES

If the file f1 contains the following text:

```
This is a file.
This is the first of three files.
This is not the last file.
```

and the file f2 contains:

```
This is a file.
This is the second of three files.
This is not the last file.
```

and the file f3 contains:

```
This is a file.
This is the third of three files.
This is the last file.
```

then the command:

```
diff3 f1 f2 f3
```

will return:

```
====
1:2,3c
   This is the first of three files.
   This is not the last file.
2:2,3c
   This is the second of three files.
   This is not the last file.
3:2,3c
   This is the third of three files.
   This is the last file
```

LIMITATIONS

Text lines that consist of a single . will defeat -e.

The diff3 command won't work on files larger than 64K bytes.

FILES

/tmp/d3*
: Temporary files

/usr/bin/diff3
: Executable file

/usr/lib/diff3prog
: Executable file

SEE ALSO

bdiff(1), cmp(1), diff(1)

NAME

diffmk — marks the differences between two files

SYNOPSIS

diffmk [-] *file1 file2 file3*

ARGUMENTS

- Causes diffmk to read *file1* from the standard input.

file1
Specifies the older version of a file.

file2
Specifies the newer version of a file.

file3
Specifies the file that is generated by diffmk, which contains the lines of *file2* plus inserted formatter ''change mark'' requests.

DESCRIPTION

diffmk compares two versions of a file and creates a third file that includes ''change mark'' requests (.mc) for nroff or troff.

When *file3* is formatted, changed, or inserted, text is shown by a pipe (|) at the right margin of each line. The position of deleted text is shown by a single asterisk (*).

If anyone is so inclined, diffmk can be used to produce listings of C (or other) programs with changes marked.

EXAMPLES

A typical command line for such use is:

```
diffmk old.c new.c tmp; nroff macs tmp | lp
```

where the file macs contains

```
.pl 1
.ll 77
.nf
.eo
.nc
```

The .ll request might specify a different line length, depending on the nature of the program being printed. The .eo and .nc requests are probably needed only for C programs.

LIMITATIONS

Aesthetic considerations may dictate manual adjustment of some output.

File differences involving only formatting requests may produce undesirable output. For example, replacing .sp by .sp 2 produces a ''change mark'' on the preceding or following line of output.

FILES

/usr/bin/diffmk
Executable file

SEE ALSO

bdiff(1), cmp(1), diff(1), diff3(1), nroff(1), troff(1)

NAME

dircmp — compares the contents of two directories

SYNOPSIS

dircmp [-d] [-s] [-w*n*] *dir1 dir2*

ARGUMENTS

-d Compares the contents of files with the same name in both directories and outputs a list telling what must be changed in the two files to bring them into agreement. The list format is described in diff.

dir1 Specifies the directory that will be compared with *dir2*.

dir2 Specifies the directory that will be compared with *dir1*.

-s Suppresses messages about identical files.

-w*n*
Changes the width of the output line to *n* characters. The default width is 72.

DESCRIPTION

dircmp examines *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Listings of files that are unique to each directory are generated for all of the options. If no option is entered, a list is output indicating whether the filenames common to both directories have the same contents.

EXAMPLES

The command:

```
dircmp d1 d2
```

will show the differences between the directories d1 and d2.

FILES

/bin/dircmp
Executable file

SEE ALSO

bdiff(1), cmp(1), diff(1), diff3(1), diffmk(1)

See basename(1)

NAME

dis — produces an assembly language listing for a specified file

SYNOPSIS

dis [-d *sec*] [-da *sec*] [-F *function*] [-l *string*] [-L] [-o] [-t *sec*] [-V] *file...*

ARGUMENTS

-d *sec*
: Disassembles the named section as .data, printing the offset of the data from the beginning of the section.

-da *sec*
: Disassembles the named section as .data, printing the actual address of the data.

file Specifies the file that is to be disassembled.

-F *function*
: Causes only those named functions from each user-supplied filename to be disassembled.

-l *string*
: Disassembles the library file specified as *string*. For example, one would issue the command

```
dis -l x -l z
```

to disassemble libx.a and libz.a. All libraries are assumed to be in /lib.

-L Invokes a lookup of C source labels in the symbol table for subsequent printing.

-o Prints numbers in octal. The default is hexadecimal.

-t *sec*
: Disassembles the named section as .text.

-V Writes the version number of the disassembler to standard error.

DESCRIPTION

dis (disassembler) produces an assembly language listing of each of its object *file* arguments. The listing includes assembly statements and the binary that produced those statements.

If the -d, -da, or -t options are specified, only those named sections from each user-supplied filename are disassembled. Otherwise, all sections containing text are disassembled.

On output, a number enclosed in brackets at the beginning of a line, such as [5], means that dis has reached the point in the assembly code where a C language line (numbered as stated) begins. If a breakpoint is placed there

using sdb/adb, the debugger used will stop on a C line. An expression such as <40> in the operand field, following a relative displacement for control transfer instructions, is the computed address within the section to which control will be transferred. A C function name will appear in the first column, followed by ().

STATUS MESSAGES AND VALUES

The self-explanatory messages indicate errors in the command line or problems encountered with the specified files.

FILES

/bin/dis
: Executable file

SEE ALSO

as(1), cc(1), ld(1), strings(1)

See enable(1)

NAME

domainname — sets or displays the name of the Network Information Service (NIS) domain

SYNOPSIS

domainname [*domain-name*]

ARGUMENTS

domain-name

Specifies the domain name for this system. Only a user who is logged in as root can specify a *domain-name* argument. In actual practice, the domain name is set by the script /etc/sysinitrc, which runs domainname with the value that is stored in the second field of the file /etc/HOSTNAME as the value of the *domain-name* argument.

DESCRIPTION

domainname, when run without the *domain-name* argument, displays the name of the domain to which this system belongs. The term ''domain'' is used to refer to a group of hosts that use the same NIS database to provide NIS service.

To make a permanent change to the domain name, log in as root, edit the second field of /etc/HOSTNAME, and restart the system to reinitialize the NIS daemons. Changing the domain name, especially if the system is a master or a slave NIS server, should be done with caution.

FILES

/bin/domainname

Executable file

SEE ALSO

ypinit(1M) in *A/UX System Administrator's Reference*

Chapter 4, ''Setting Up the Network Information Service,'' in *A/UX Network System Administration*

NAME

du — summarizes disk usage

SYNOPSIS

du [-a[l]] [-r] [-s] [*files*]

ARGUMENTS

-a Causes an entry to be generated for each file. If this option is not specified, an entry is generated for each directory only.

files Specifies the files to be summarized. If this argument is not given, a dot (.) is used.

-l Causes additional information for symbolic links to be displayed, indicating the full path to the file that each symbolic link references. Note that the size shown is the block size for the symbolic link itself, not that of the file that it references.

-r Causes du to generate messages about directories that cannot be read, files that cannot be opened, and so on.

-s Causes only the grand total (for each of the specified files) to be given.

DESCRIPTION

du displays the number of blocks contained in all files (recursively) and directories within each directory and file specified by the *files* argument. The default system size for physical blocks is 512 bytes. The block count includes the indirect blocks of the file.

The du command is normally silent about directories that cannot be read, files that cannot be opened, and so on.

A file with two or more links is counted only once.

EXAMPLES

The following command produces a count of the number of (512-byte) blocks in each of the directories.

```
du dir1 dir2
```

To see how many blocks are in each file, you must use the -a option.

LIMITATIONS

If you do not use the -a option, nondirectories given as arguments are not listed.

If there are too many distinct linked files, du will count the excess files more than once.

Files with holes in them will cause an incorrect block count.

FILES

`/bin/du`
Executable file

SEE ALSO

`df`(1)

NAME

dump — stores (saves) selected parts of an object file

SYNOPSIS

dump [[-a] [-c] [-f] [-g] [-h] [-l] [-o] [-r] [-s] [-t] [-z *name*]] [[-d *number*] [+d *number*] [-n *name*] [-p] [-t *index*] [+t *index*] [-u] [-z *name*, *number*] [+z *name*]] *file...*

dump [[-a] [-c] [-f] [-g] [-h] [-l] [-r] [-t] [-z *name*]] [[-d *number*] [+d *number*] [-n *name*] [-p] [-t *index*] [+t *index*] [-u] [-v] [-z *name*, *number*] [+z *name*]] *file...*

ARGUMENTS

-a Dumps the archive header of each member of each archive file argument.

-c Dumps the string table.

-d *number*
Dumps the section number or range of sections starting at *number* and ending either at the last section number or *number* specified by +d.

+d *number*
Dumps sections in the range either beginning with first section or beginning with section specified by -d.

-f Dumps each file header.

file Specifies the object file or an archive of object files to be saved.

-g Dumps the global symbols in the symbol table of a version 6.0 archive.

-h Dumps section headers.

-l Dumps line number information.

-n *name*
Dumps information pertaining only to the named entity. This *modifier* applies to the -h, -s, -r, -l, and -t options.

-o Dumps each optional header.

-p Suppresses the printing of the headers.

-r Dumps relocation information.

-s Dumps section contents.

-t Dumps symbol table entries.

-t *index*
Dumps only the indexed symbol table entry. When the -t option is used in conjunction with the +t option, it specifies a range of symbol table entries.

`+t` *index*
Dumps the symbol table entries in the range ending with the indexed entry. The range begins at the first symbol table entry or at the entry specified by the `-t` option.

`-u` Underlines the name of the file for emphasis.

`-v` Dumps information in symbolic representation rather than numeric (for example, `C_STATIC` instead of `0X02`). This *modifier* can be used with all the preceding options except the `-s` and `-o` options.

`-z` *name*
Dumps line number entries for the named function.

`-z` *name*`,` *number*
Dumps line number entry or range of line numbers starting at *number* for the named function.

`+z` *name*
Dumps line numbers starting at either function *name* or *number* specified by the `-z` option up to *number* specified by this option.

DESCRIPTION

`dump` stores (or saves) selected parts of each of its object *file* arguments.

This command accepts both object files and archives of object files. It processes each file argument according to one or more of the options.

Blanks separating an option and its modifier are optional. The comma separating the name from the number modifying the `-z` option may be replaced by a blank.

The `dump` command attempts to format the information it dumps in a meaningful way, printing certain information in character, hex, octal, or decimal representation, as appropriate.

FILES

`/bin/dump`
Executable file

SEE ALSO

`as`(1), `dis`(1), `od`(1), `nm`(1), `strings`(1)

`a.out`(4), `ar`(4) in *A/UX Programmer's Reference*

See ex(1)

NAME

echo — echoes its arguments

SYNOPSIS

echo [*arg*]...

ARGUMENTS

arg Specifies the argument entered by the user which will be echoed.

DESCRIPTION

echo writes its arguments separated by blanks and terminated by a newline on the standard output. It also understands C-like escape conventions; beware of conflicts with the shell's use of \:

\b backspace

\c print line without newline

\f form-feed

\n newline

\r carriage return

\t tab

\v vertical tab

\\ backslash

n the 8-bit character whose ASCII code is the 1-, 2- or 3-digit octal number *n*, which must start with a zero.

The echo command is useful for producing messages in command files and for sending known data into a pipe. A version of echo is built into the Bourne shell (sh(1)). Similar versions are also built into ksh(1) and csh(1).

EXAMPLES

The command:

```
echo curmudgeon
```

simply responds

```
curmudgeon
```

on the standard output.

FILES

/bin/echo
Executable file

SEE ALSO

csh(1), ksh(1), sh(1)

NAME

ed, red — edit text

SYNOPSIS

ed [-] [-p *string*] [-x] [*file*]

red [-] [-p *string*] [-x] [*file*]

ARGUMENTS

- Suppresses the printing of character counts by e, r, and w commands, of diagnostics from e and q commands, and of the ! prompt after a !*shell command*.

file Causes ed to simulate an *e* command on the named file; that is to say, the file is read into ed's buffer so that it can be edited.

-p *string*
Allows the user to specify a prompt string. The string must be enclosed in double quotes.

-x Causes an X command to be simulated first to handle an encrypted file. This option and the editor command X are not implemented in the international distribution.

DESCRIPTION

ed is the standard text editor. The ed command operates on a copy of the file it is editing; changes made to the copy have no effect on the file until a w (write) command is given. The copy of the text being edited resides in a temporary file called the ''buffer.'' There is only one buffer.

red is a restricted version of ed. It will allow editing of files only in the current directory. It prohibits executing shell commands via !*shell command*. Attempts to bypass these restrictions result in the error message:

```
restricted shell
```

Both ed and red support the fspec(4) formatting capability. After including a format specification as the first line of *file* and invoking ed with your terminal in stty -tabs or stty tab3 mode (see stty(1)), the specified tab stops will be used automatically when scanning *file*. For example, if the first line of a file contained:

```
<:t5,10,15 s72:>
```

tab stops would be set at columns 5, 10, and 15, and a maximum line length of 72 would be imposed.

Note: While entering text, tab characters, when typed, are expanded to every eighth column, as is the default.

Commands to ed have a simple and regular structure: zero, one, or two addresses followed by a single-character command, followed by any applicable parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses very often can be omitted.

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

The ed command supports a limited form of ''regular expression'' (RE) notation; regular expressions are used in addresses to specify lines and in some commands (for example, *s*) to specify portions of a line that are to be substituted. A regular expression specifies a set of character strings. A member of this set of strings is said to be ''matched'' by the RE. The REs allowed by ed are constructed as follows:

The following one-character REs match a single character:

1.1 An ordinary character (*not* one of those discussed in 1.2 next) is a one-character RE that matches itself.

1.2 A backslash (\ ) followed by any special character is a one-character RE that matches the special character itself. The special characters are:

a. . , *, [, and \ (period, asterisk, left square bracket, and backslash, respectively), which are always special, except when they appear within square brackets ([ ]; see paragraph 1.4).

b. ^ (circumflex), which is special at the beginning of an entire RE (see paragraphs 3.1 and 3.2), or when it immediately follows the left of a pair of square brackets ([ ]) (see paragraph 1.4).

c. $ (currency symbol), which is special at the end of an entire RE (see paragraph 3.2).

d. The character used to bound (that is, delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the g command.)

1.3 A period (.) is a one-character RE that matches any character except newline.

1.4 A nonempty string of characters enclosed in square brackets ([ ]) is a one-character RE that matches ''any one'' character in that string. If, however, the first character of the string is a circumflex (^), the one-character RE matches any character except newline and the

remaining characters in the string. The ^ has this special meaning only if it occurs first in the string. The minus (-) may be used to indicate a range of consecutive ASCII characters; for example, [0-9] is equivalent to [0123456789]. The - loses this special meaning if it occurs first (after an initial ^, if any) or last in the string. The right square bracket (]) does not terminate such a string when it is the first character within it (after an initial ^, if any); for example, `[]a-f]` matches either a right square bracket (]) or one of the letters `a` through `f`, inclusive. The four characters, listed in paragraph 1.2, item a., stand for themselves within such a string of characters.

The following rules may be used to construct REs from one-character REs:

2.1 A one-character RE is an RE that matches whatever the one-character RE matches.

2.2 A one-character RE followed by an asterisk (*) is a RE that matches zero or more occurrences of the one-character RE. If there is any choice, the longest left-most string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m,*\}, or \{*m,n*\} is an RE that matches a range of occurrences of the one-character RE. The values of *m* and *n* must be non-negative integers less than 256:

\{*m*\} matches exactly *m* occurrences;

\{*m,*\} matches "at least" *m* occurrences;

\{*m,n*\} matches "any number" of occurrences between *m* and *n* inclusive.

Whenever a choice exists, the RE matches as many occurrences as possible.

2.4 The concatenation of REs is a RE that matches the concatenation of the strings matched by each component of the RE.

2.5 An RE enclosed between the character sequences \(and \) is an RE that matches whatever the unadorned RE matches.

2.6 The expression *n* matches the same string of characters as was matched by an expression enclosed between \(and \) earlier in the same RE. Here *n* is a digit; the sub-expression specified is that beginning with the *n*th occurrence of \(counting from the left. For example, the expression ^\(.*\)\1$ matches a line consisting of two repeated appearances of the same string.

Finally, an entire RE may be constrained to match only an initial segment or final segment of a line (or both).

3.1 A caret (^) at the beginning of an entire RE constrains that RE to match an initial segment of a line.

3.2 A currency symbol ($) at the end of an entire RE constrains that RE to match a final segment of a line.

The construction ^*entire RE*$ constrains the entire RE to match the entire line.

The null RE (for example, //) is equivalent to the last RE encountered. See the paragraph before the ''Files'' section, at the end of this manual page.

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

1. The character . addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A decimal number *n* addresses the *n*th line of the buffer.
4. *'x* addresses the line marked with the mark name character *x*, which must be a lowercase letter. Lines are marked with the *k* command. If *x* was not used to mark a line, *'x* addresses line 0.
5. An RE enclosed by slashes (/) addresses the first line found by searching forward from the line following the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the beginning of the buffer and continues up to and includes the current line, so that the entire buffer is searched. See the paragraph before the ''Files'' section at the end of this manual page.
6. An RE enclosed in question marks (?) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and includes the current line. See also the last paragraph before the ''Files'' section at the end of this manual page.
7. An address followed by a plus sign (+) or a minus sign (-) followed by a decimal number specifies that address plus (respectively, minus) the indicated number of lines. The plus sign may be omitted.
8. If an address begins with + or -, the addition or subtraction is taken with respect to the current line; for example, -5 is understood to mean .-5.

9. If an address ends with + or -, then 1 is added to or subtracted from the address, respectively. As a consequence of this rule, and of the preceding rule, the address - refers to the line preceding the current line. (To maintain compatibility with earlier versions of the editor, the character ^ in addresses is entirely equivalent to -.) Moreover, trailing + and - characters have a cumulative effect, so -- refers to the current line less 2.
10. For convenience, a comma (,) stands for the address pair 1,$, while a semicolon (;) stands for the pair .,$.

Commands may require zero, one, or two addresses. Commands that require no addresses regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when an insufficient number of addresses is given; if more addresses are given than such a command requires, the last one(s) are used.

Typically, addresses are separated from each other by commas (,). They may also be separated by semicolons (;). In the latter case, the current line (.) is set to the first address, and only then is the second address calculated. This feature can be used to determine the starting line for forward and backward searches (see rules 5. and 6. preceding). The second address of any two-address sequence must correspond to a line that follows, in the buffer, the line corresponding to the first address.

In the following list of ed commands, the default addresses are shown in parentheses. The parentheses are not part of the address; they show that the given addresses are the default.

It is generally illegal for more than one command to appear on a line. Any command (except e, f, r, or w) may be suffixed by l, n or p, however, in which case the current line is either listed, numbered or printed, respectively, as discussed under the l, n and p commands.

(.)a
text
.
The append command reads the given text and appends it after the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. Address 0 is legal for this command; it causes the appended text to be placed at the beginning of the buffer. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(.)c
text
.
The change command deletes the addressed lines, then accepts input text that replaces these lines; . is left at the last line input, or, if there were none, at the first line that was not deleted.

`(.,.)d`
The delete command deletes the addressed lines from the buffer. The line after the last line deleted becomes the current line; if the lines deleted were originally at the end of the buffer, the new last line becomes the current line.

`e` *file*
The edit command causes the entire contents of the buffer to be deleted, and then the named file to be read in; `.` is set to the last line of the buffer. If no filename is given, the currently-remembered filename, if any, is used (see the `f` command). The number of characters read is typed; *file* is remembered for possible use as a default filename in subsequent `e`, `r`, and `w` commands. If *file* is replaced by `!`, the rest of the line is taken to be a shell (`sh`(1)) command whose output is to be read. Such a shell command is not remembered as the current filename. See also the ''Status Message and Values'' section, later in the manual page.

`E` *file*
The `E` command is like `e`, except that the editor does not check to see if any changes have been made to the buffer since the last `w` command.

`f` *file*
If *file* is given, this command changes the currently-remembered filename to *file*; otherwise, it prints the currently-remembered filename.

`(1,$)g/`*RE*`/`*command list*
In the global command, the first step is to mark every line that matches the given RE. Then, for every such line, the given *command list* is executed with `.` initially set to that line. A single command or the first of a list of commands appears on the same line as the global command. All lines of a multi-line list except the last line must be ended with a `\`; `a`, `i`, and `c` commands and associated input are permitted. The `.` terminating input mode may be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the `p` command.

`(1,$)G/`*RE*`/`
In the interactive global command, the first step is to mark every line that matches the given RE. Then, for every such line, that line is printed, `.` is changed to that line, and any one command (other than one of the `a`, `c`, `i`, `g`, `G`, `v`, and `V` commands) may be input and is executed. After the execution of that command, the next marked line is printed, and so on; a newline acts as a null command; an `&` causes the re-execution of the most recent command executed within the

current invocation of G. Note that the commands input as part of the execution of the G command may address and affect any lines in the buffer. The G command can be terminated by an interrupt signal (ASCII DELETE or BREAK). A command that causes an error terminates the G command.

h The help command gives a short error message that explains the reason for the most recent ? diagnostic.

H The Help command causes ed to enter a mode in which error messages are printed for all subsequent ? diagnostics. It will also explain the previous ? if there was one. The H command alternately turns this mode on and off; it is initially off.

(.)i
text
.

The insert command inserts the given text before the addressed line; . is left at the last inserted line, or, if there were none, at the addressed line. This command differs from the a command only in the placement of the input text. Address 0 is not legal for this command. The maximum number of characters that may be entered from a terminal is 256 per line (including the newline character).

(. , .+1)j

The join command joins contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command does nothing.

(.)k*x*

The mark command marks the addressed line with name *x*, which must be a lowercase letter. The address *'x* then addresses this line; . is unchanged.

(. , .)l

The list command lists the addressed lines in an unambiguous way: A few nonprinting characters (for example, *tab, backspace*) are represented by mnemonic overstrikes. All other nonprinting characters are printed in octal, and long lines are folded. An l command may be appended to any command other than e, f, r, or w.

(. , .)m*a*

Move addressed line(s) to after the line addressed by a. Address 0 is legal for a and causes the addressed line(s) to be moved to the beginning of the file. It is an error if address a falls within the range of moved lines; . is left at the last line moved.

(. , .)n

The print command prints the addressed lines, preceding each line by its line number and a tab character; . is left at the last line printed.

The n command may be appended to any command other than e, f, r, or w.

(. , .)p
The print command prints the addressed lines; . is left at the last line printed. The p command may be appended to any command other than e, f, r, or w. For example, dp deletes the current line and prints the new current line.

P The editor will prompt with a * for all subsequent commands. The P command alternately turns this mode on and off; it is initially off.

q The q command exits. ed No automatic write of a file is done (but see the ''Status Messages and Values'' section later in the manual page.

Q The Q command exits ed without checking if changes have been made in the buffer since the last w command.

($)r *file*
The read command reads in the given file after the addressed line. If no filename is given, the currently-remembered filename, if any, is used (see e and f commands). The currently-remembered filename is not changed unless *file* is the very first filename mentioned since ed was invoked. Address 0 is legal for r and causes the file to be read at the beginning of the buffer. If the read is successful, the number of characters read is typed; . is set to the last line read in. If *file* is replaced by !, the rest of the line is taken to be a shell (sh(1)) command whose output is to be read. For example, $r !ls appends the current directory to the end of the file being edited. Such a shell command is not remembered as the current filename.

(. , .) s/*RE*/*replacement*/ or
(. , .) s/*RE*/*replacement*/g or
(. , .) s/*RE*/*replacement*/*n*
These commands search each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (nonoverlapped) matched strings are replaced by the *replacement* if the global replacement indicator g appears after the command. If the global indicator does not appear, only the first occurrence of the matched string is replaced. Sometimes substitution of an RE results in the last (or only) affected line being printed out. This occurs only when substitution is not global or of an *n*th occurrence. If a number *n* appears after the command, only the *n*th occurrence of the matched string on each addressed line is replaced. It is an error for the substitution to fail on *all* addressed lines. Any character other than space or newline may be used instead of / to delimit the RE and the

replacement; . is left at the last line on which a substitution occurred. See the paragraph before the ''Files'' section at the end of this manual page.

An ampersand (&) appearing in the *replacement* is replaced by the string matching the RE on the current line. The special meaning of & in this context may be suppressed by preceding it with a \. As a more general feature, the characters *n*, where *n* is a digit, are replaced by the text matched by the *n*th regular subexpression of the specified RE enclosed between \(and \). When nested parenthesized subexpressions are present, *n* is determined by counting occurrences of \(starting from the left. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command is used as the *replacement* in the current substitute command. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a \.

A line may be split by substituting a newline character into it. The newline in the *replacement* must be escaped by preceding it with a \. Such substitution cannot be done as part of a g or v command list.

(.,.)t*a*
This command is similar to the move (m) command, except that a copy of the addressed lines is placed after address *a* (which may be 0); . is left at the last line of the copy.

u The u command undoes the most recent command that modified anything in the buffer, namely the most recent a, c, d, g, i, j, m, r, s, t, v, G, or V command.

(1,$)v/*RE*/*command list*
This command is the same as the global command g, except that the *command list* is executed with . initially set to every line that does not match the RE.

(1,$)V/*RE*/
This command is the same as the interactive global command G except that the lines that are marked during the first step are those that do not match the RE.

(1,$)w *file*
This command writes the addressed lines into the named file. If the file does not exist, it is created with mode 666 (readable and writable by everyone), unless your umask setting (see sh(1)) dictates otherwise. The currently-remembered filename is not changed unless *file* is the very first filename mentioned since ed was invoked. If no filename is given, the currently-remembered filename, if any, is used (see e and f commands); . is unchanged. If the command is

successful, the number of characters written is typed. If *file* is replaced by `!`, the rest of the line is taken to be a shell (`sh`(1)) command whose standard input is the addressed lines. Such a shell command is not remembered as the current filename.

`X` A key string is demanded from the standard input. Subsequent `e`, `r`, and `w` commands will encrypt and decrypt the text with this key by the algorithm of `crypt`(1). An explicitly empty key turns off encryption. The encryption scheme used here is not secure.

`( $ )=`
The line number of the addressed line is typed; address 0 is legal for this command. `.` is unchanged by this command.

`!`*shell command*
The remainder of the line after the `!` is sent to the system shell (`sh`(1)) to be interpreted as a command. Within the text of that command, the unescaped character `%` is replaced with the remembered filename; if a `!` appears as the first character of the shell command, it is replaced with the text of the previous shell command. Thus, `!!` will repeat the last shell command. If any expansion is performed, the expanded line is echoed; `.` is unchanged.

`( .+1 )` newline
An address alone on a line causes the addressed line to be printed. A newline alone is equivalent to `.+1p`; it is useful for stepping forward through the buffer.

If an interrupt signal (ASCII or CONTROL-c is sent, `ed` prints a `?` and returns to its command level.

Some size limitations: 512 characters per line, 256 characters per global command list, 64 characters per filename, and 128K characters in the buffer. The limit on the number of lines depends on the amount of user memory; each line takes 1 word.

When reading a file, `ed` discards ASCII NUL characters and all characters after the last newline. Files (for example, `a.out`) that contain characters not in the ASCII set (bit 8 on) cannot be edited by `ed`.

If the closing delimiter of an RE or of a replacement string (for example, `/`) would be the last character before a newline, that delimiter may be omitted, in which case the addressed line is printed. The following pairs of commands are equivalent:

```
s/s1/s2      s/s1/s2/p
g/s1  g/s1/p
?s1   ?s1?
```

EXAMPLES

The command:

```
ed text
```

invokes the editor with the file named `text`. For further examples, see "Using `ed`" in *A/UX Text Editing Tools*.

STATUS MESSAGES AND VALUES

`?` for command errors.

`?`*file*
for an inaccessible file.
(use the `h` and `H` commands for detailed explanations).

If changes have been made in the buffer since the last `w` command that wrote the entire buffer, `ed` warns the user if an attempt is made to destroy `ed`'s buffer via the `e` or `q` commands. It prints `?` and allows one to continue editing. A second `e` or `q` command will take effect at any time, provided no further changes have been made to the file. The `-` command-line option inhibits this feature.

NOTES

The `!` command and the `!` escape from the `e`, `r`, and `w` commands cannot be used if the the editor is invoked from a restricted shell (see `sh`(1)).

The sequence `\n` in an RE does not match a newline character.

The `l` command mishandles interrupts.

Files encrypted directly with the `crypt`(1) command with the null key cannot be edited.

Characters are masked to 7 bits on input.

If the editor input is coming from a command file (that is, `ed` *file* `<` *ed-cmd-file*), the editor will exit at the first failure of a command that is in the command file.

FILES

`/bin/ed`
Executable file

`/bin red`
Executable file

`/tmp/e#`
Temporary file # is the process number

`ed.hup`
File containing work when the terminal is hung up

SEE ALSO

`crypt`(1), `ex`(1), `grep`(1), `sed`(1), `sh`(1), `stty`(1), `vi`(1)

`fspec`(4), `regexp`(5) in *A/UX Programmer's Reference*

"Using `ed`" in *A/UX Text Editing Tools*

edit(1)

See ex(1)

NAME

efl — invokes the Extended Fortran Language

SYNOPSIS

efl [-#] [-C] [-w] [*file*]...

ARGUMENTS

-# Suppresses comments in the generated program, and the default option.

-C Causes comments to be included in the generated program.

file Specifies the file to be compiled.

-w Suppresses warning messages.

DESCRIPTION

efl compiles a program written in the efl language into clean Fortran on the standard output. The efl command provides the C-like control constructs similar to Ratfor:

statement grouping with braces.

decision-making:
if, if-else, and select-case (also known as switch-case);
while, for, Fortran do, repeat, and repeat ... until loops;
multi-level break and next.

The efl command has C-like data structures, for example:

```
struct
      {
      integer flags(3)
      character(8) name
      long real coords(2)
      } table(100)
```

The language offers generic functions, assignment operators (+=, &=, and so on.), and sequentially evaluated logical operators (&& and ||). There is a uniform input/output syntax:

```
write(6,x,y:f(7,2), do i=1,10 { a(i,j),z.b(i) })
```

The efl command also provides some syntactic ''sugar'':

free-form input:
multiple statements per line; automatic continuation; statement label names (not just numbers).

comments:

```
# this is a comment.
```

translation of relational and logical operators:
>, >=, &, and so on, become .GT., .GE., .AND., and so on,

return expression to caller from function:

```
return (expression)
```

defines:
`define` *name replacement*

includes:
`include` *file*

An argument with an embedded = (equal sign) sets an `efl` option as if it had appeared in an `option` statement at the start of the program. Many options are described in the reference manual. A set of defaults for a particular target machine may be selected by one of the choices: `system=unix`, `system=gcos`, or `system=cray`. The default setting of the `system` option is the same as the machine on which the compiler is running. Other specific options determine the style of input/output, error handling, continuation conventions, the number of characters packed per word, and default formats.

The `efl` command is best used with `f77`(1).

EXAMPLES

The command sequence:

```
efl prog.for > prog.f
f77 prog.f -o prog
```

will process the program `prog.for` through `efl` and then run the `f77`(1) compiler on the output from `efl`, generating an executable file named `prog`.

FILES

`/usr/bin/efl`
Executable file

SEE ALSO

`cc`(1), `f77`(1)

"EFL Reference," in *A/UX Programming Languages and Tools, Volume 1*

See grep(1)

NAME

eject — ejects a diskette from the drive

SYNOPSIS

eject [0] [1] [/dev/rdsk/*name*]

ARGUMENTS

0 Specifies the diskette in disk drive zero (0) to be ejected.

1 Specifies the diskette in disk drive one (1) to be ejected.

/dev/rdsk/*name*
Specifies the full pathname of the character special file for the device. If the device is not given, the device 0 is assumed.

DESCRIPTION

eject causes a floppy diskette drive (see fd(7)) to eject an inserted diskette. One of three arguments may be included on the command line.

FILES

/bin/eject
Executable file

/dev/rdsk/c8d?s0
Tape device

SEE ALSO

fd(7) in *A/UX Programmer's Reference*

NAME

enable, disable — enable or disable LP printers

SYNOPSIS

enable *printers*

disable [-c] [-r[*reason*]] *printers*

ARGUMENTS

-c Cancels any requests that are currently printing on any of the designated printers.

printers
Specifies the printers to be activated or deactivated.

-r[*reason*]
Associates a *reason* with the deactivation of the printers. This reason applies to all printers mentioned up to the next -r option. If the -r option is not present or the -r option is given without a reason, then a default reason will be used. *reason* is reported by lpstat.

DESCRIPTION

enable activates the named *printers*, enabling them to print requests taken by lp(1). Use lpstat(1) to find the status of printers.

disable deactivates the named *printers*, disabling them from printing requests taken by lp(1). By default, any requests that are currently printing on the designated printers will be reprinted in their entirety either on the same printer or on another member of the same class. Use lpstat(1) to find the status of printers.

FILES

/usr/bin/enable
Executable file
/usr/bin/disable
Executable file
/usr/spool/lp/*
Spooler files

SEE ALSO

lp(1), lpstat(1)

A/UX Local System Administration

NAME

enscript — converts text files to POSTSCRIPT format for printing

SYNOPSIS

enscript [-1] [-2] [-b*header*] [-B] [-f*font*] [-F*hfont*] [-g] [-G] [-h] [-k] [-K] [-l] [-L*lines*] [-m] [-o] [-p*out*] [-q] [-r] [-R] [[-#*n*] [-C*class*] [-J*name*] [-P*printer*]] [*files*]

enscript [-1] [-2] [-b*header*] [-B] [-f*font*] [-F*hfont*] [-g] [-G] [-h] [-k] [-K] [-l] [-L*lines*] [-m] [-o] [-p*out*] [-q] [-r] [-R] [[-d*dest*] [-n*n*] [-t*title*] [-w] [*files*]

ARGUMENTS

-#*n*
: Produces *n* copies. The default is one.

-1 Sets the text in one column (the default).

-2 Sets the text in two columns.

-b*header*
: Sets the string to be used for page headings to *header*. The default header is constructed from the filename, its last modification date, and a page number.

-B Omits page headings.

-C*class*
: Sets the job classification for use on the burst page.

-d*dest*
: Sends the output to the named printer or printer class.

-f*font*
: Sets the font to be used for the body of each page. Defaults to Courier10, unless two-column rotated mode is used, in which case it defaults to Courier7.

files Specifies the text files to be converted.

-F*hfont*
: Sets the font to be used for page headings. Defaults to Courier-Bold10.

-g Enables the printing of files containing nonprinting characters. Any file with more than a small number of nonprinting characters is suspected of being garbage and is not printed unless this option is used.

-G Prints in gaudy mode; causes page headings, dates, page numbers to be printed in a flashy style, at some slight performance expense.

`-h` Suppresses printing of a job burst page.

`-J`*name*
Sets the job name for use on the burst page. Otherwise, the name of the first input file is used.

`-k` Enables page prefeed (if the printer supports it). This allows simple documents (for example, program listings in one font) to print somewhat faster by keeping the printer running between pages.

`-K` Disables page prefeed (the default).

`-l` Simulates a line printer; makes pages 66 lines long and omit headers.

`-L`*lines*
Sets the maximum number of lines to output on a page. The `enscript` command usually computes how many to put on a page based on point size, and may put fewer per page than requested by *lines*.

`-m` Sends mail via `mail`(1) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

`-n`*n*
Produces *n* copies. The default is one.

`-o` Lists any missing characters whenever characters cannot be found within a specified font.

`-p`*out*
Writes the POSTSCRIPT file to the named file rather than spooling it for printing. As a special case, `-p` will send the POSTSCRIPT to the standard output.

`-P`*printer*
Sends the output to the named printer.

`-q` Suppresses the status messages. The `enscript` command won't report about pages, destination, omitted characters, and so forth. Fatal errors are still reported to the standard error output.

`-r` Rotates the output 90 degrees (landscape mode). This is good for output that requires a wide page or for program listings when used in conjunction with the `-2` option.

```
enscript -2r files
```

is a nice way to get program listings.

`-R` Does not rotate. This option is also known as portrait mode (the default).

-t*title*
Sets the job title for use on the burst page.

-w Writes a status message to the user's terminal after files have been printed.

DESCRIPTION

enscript reads plain text files, converts them to POSTSCRIPT format, and spools them for printing on a POSTSCRIPT printer. Fonts, headings, and limited formatting and spooling options may be specified. By default, the print spooler used by enscript is the Berkeley spooler, lpr. The environment variable SPOOLER may be set to specify the System V spooler, lp.

For example

```
enscript -Paleph boring.txt
```

processes the file called boring.txt for POSTSCRIPT printing, sending the output to the printer aleph.

```
enscript -2r boring.c
```

prints a two-up landscape listing of the file called boring.c on the default printer.

Font specifications have two parts: A font name that POSTSCRIPT recognizes (for example, Times-Roman, Times-Roman BoldItalic , Helvetica, Courier), and a point size (1 point=1/72 inch). So, Courier-Bold8 is 8-point Courier Bold, Helvetica12 is 12-point Helvetica.

The environment variable ENSCRIPT may be used to specify defaults. The value of ENSCRIPT is parsed as a string of arguments before the arguments that appear on the command line. For example:

```
ENSCRIPT='-fTimes-Roman8'
```

sets your default body font to 8-point Times Roman.

The -#*n*, -C*class*, -J*name*, and -P*printer* spooler options are passed to the lpr command.

The -d*dest*, -n*n*, -t*title*, and -w spooler options are passed to the lp command.

Environment Variables

The following environmental variables may be used in conjunction with enscript:

SPOOLER
The name of the print spooler, lpr or lp, for enscript to use. If SPOOLER is not set, enscript will spool to lpr.

ENSCRIPT
String of options to be used by enscript.

PSLIBDIR
Pathname of a directory to use instead of /usr/lib/ps for enscript prologue and font metric files.

PSTEMPDIR
Pathname of temporary directory to use instead of XPSTEMDIRX of spooled temporary files.

PRINTER
The name of a printer (as in the -P option) for lpr to use. If no -P option is specified, lpr will use this printer. If neither -P nor PRINTER is set, enscript will spool to a printer named "PostScript". This environment variable has no effect on the spooler lp.

LPDEST
The name of a printer for lp to use. If LPDEST is not set, enscript will spool to a printer class named PostScript. This environment variable has no effect on the spooler lpr.

Features

Options and the ENSCRIPT environment string are parsed in getopt(3) fashion.

LIMITATIONS

Long lines are truncated. Because printer margins vary, line truncation may be off. There should be a "wrap" option and multiple (truncated or wrapped) columns.

NOTES

POSTSCRIPT is a trademark of Adobe Systems Incorporated.
Times and Helvetica are registered trademarks of Linotype.

FILES

/usr/bin/enscript
Executable file
/usr/lib/ps/*.afm
Font metrics files
/usr/lib/ps/enscript.pro
Prologue file for enscript files

SEE ALSO

lp(1), lpr(1), lprm(1), lpstat(1), pr(1)

getopt(3C) in *A/UX Programmer's Reference*

NAME

env — sets the environment for command execution

SYNOPSIS

env [-] [*name=value*]... [*command args*]

ARGUMENTS

- Causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

args
Specifies the arguments for the command that is to be executed.

command
Specifies the command to be executed. If no command is specified, the resulting environment is printed, one name-value pair per line.

name=value
Merges *name=value* into the inherited environment before the command is executed.

DESCRIPTION

env obtains the current environment, modifies it according to its arguments, then executes *command* with the modified environment.

EXAMPLES

The command:

```
env XYZ=pdq sh
```

sets XYZ to the value pdq for the duration of the command, which here is a new shell, sh.

FILES

/bin/env
Executable file

SEE ALSO

csh(1), ksh(1), sh(1)

exec(2), profile(4), environ(5) in *A/UX Programmer's Reference*

NAME

eqn, checkeq — format mathematical text for troff

SYNOPSIS

eqn [-d*xy*] [-f*n*] [-p*n*] [-s*n*] [-T*tty-type*] [-] [*file*]...

checkeq [*file*]...

ARGUMENTS

- Causes eqn to read the standard input.

-d*xy*
Sets delimiters to characters *x* and *y* with the command-line argument -d*xy* or (more commonly) with delim *xy* between .EQ and .EN. The left and right delimiters may be the same character, the dollar sign often being used as such a delimiter. Delimiters are turned off by delim off. All text that is neither between delimiters nor between .EQ and .EN is passed through untouched.

-f*n*
Specifies the font to be used. Replace *n* with the desired font.

file Specifies the file to be formatted. If this option is not given, eqn will read the standard input.

-p*n*
Specifies the point size of the equation. Replace *n* with a point size. The legal point size numbers are:

```
6  7  8  9  10  11  12  14
16  18  20  22  24  28  36
```

-s*n*
Specifies the size the equation.

-T*tty-type*
Causes eqn to prepare output for the specified output device, *tty-type*. The currently supported devices are -Taps (Autologic APS-5) and -Tpsc (POSTSCRIPT device). The default is -Tpsc.

DESCRIPTION

eqn is a troff(1) preprocessor for typesetting mathematical text on a phototypesetter. Normal usage is

eqn [*options*] *file* ... | troff [*options*] | [*typesetter*]

A line beginning with .EQ marks the start of an equation; a line beginning with .EN marks the end of an equation; troff does not alter these lines, so they may be defined in macro packages to get centering, numbering, and so forth. You may also name two characters as delimiters; eqn treats subsequent text between delimiters as input.

checkeq is a related program that reports missing or unbalanced delimiters and .EQ/.EN pairs.

Tokens within eqn are separated by spaces, tabs, newlines, braces, double quotation marks, tildes, and circumflexes. Braces ({}) are used for grouping; generally, wherever a single character such as *x* may be used, then *x* enclosed in braces may be used instead. A tilde (~) represents a full space in the output; a circumflex (^) half as much.

For full details, see "eqn Reference" in *A/UX Text Processing Tools.*

LIMITATIONS

To boldface digits, parentheses, and so forth, it is necessary to enclose them in quotation marks, as in

```
bold "12.3"
```

When you use eqn with the mm macro package, displayed equations must appear only inside displays.

See also LIMITATIONS under troff(1).

FILES

/bin/checkeq
Executable file

/bin/eqn
Executable file

SEE ALSO

mm(1), mmt(1), nroff(1), tbl(1), troff(1)

eqnchar(5), mm(5), mv(5) in *A/UX Programmer's Reference*

"eqn Reference" in *A/UX Text Processing Tools*

NAME

e, ex, edit — edit text

SYNOPSIS

ex [-] [+*command*] [-r] [-R] [-t *tag*] [-v] [-x] *file...*

e [-] [+*command*] [-r] [-R] [-t *tag*] [-v] [-x] *file...*

edit [-] [+*command*] [-r] [-R] [-t *tag*] [-v] [-x] *file...*

ARGUMENTS

- Suppresses all interactive-user feedback, as when processing editor scripts in command files.

+*command*
Indicates that the editor should begin by executing the specified command. If *command* is omitted, then it defaults to $, positioning the editor initially at the last line of the first file. Other useful commands here are scanning patterns of the form /*pat* or line numbers, for example, +100 to start at line 100.

file Specifies the file to be edited.

-r Recovers files after an editor or system crash, retrieving the last saved version of the named file. If no *file* is specified, a list of saved files will be reported.

-R Specifies the read-only mode set and prevents accidental overwriting of the file.

-t *tag*
Acts the same as an initial *tag* command, editing the file containing the *tag* and positioning the editor at its definition.

-v Uses vi rather than ex.

-x Specifies encryption mode; a key is prompted for allowing creation or editing of an encrypted file. This encryption scheme is not secure.

DESCRIPTION

ex is the root of a family of editors: edit, ex, and vi. The edit command set is a subset of the ex set, including just the basic commands, fewer magic characters, and line-based editing only. Display-based editing is the focus of vi.

If you have not used ed, or are a casual user, you will find that the editor edit is convenient for you. It avoids some of the complexities of ex, which is used mostly by systems programmers and those very familiar with ed.

e is synonymous with ex.

If you have a CRT terminal, you may wish to use a display-based editor; in this case see vi(1), which is a command that focuses on the display editing portion of ex.

Modes

Command
: Normal and initial state. Input prompted for by :. The kill character cancels partial command.

Insert
: Entered by a, i, and c. Arbitrary text may be entered. Insert is normally terminated by line having only . on it, or abnormally with an interrupt.

Visual
: Entered by vi, terminates with Q or ^\.

Command Names and Abbreviations

abbrev	ab	next	n	undo	u
append	a	number	nu	unmap	unm
args	ar	preserve	pre	version	ve
change	c	print	p	visual	vi
copy	co	put	pu	write	w
delete	d	quit	q	xit	x
edit	e	read	re	yank	ya
file	f	recover	rec	window	z
global	g	rewind	rew	escape	!
insert	i	set	se	lshift	<
join	j	shell	sh	printnext	CR
list	l	source	so	resubst	&
map	map	stop	st	rshift	>
mark	ma	substitute	s	scroll	^D
move	m	unabbrev	una		

where CR=RETURN, and ^D=CONTROL-D.

Command Addresses

n	line *n*	*/pat*	next with *pat*
.	current	?*pat*	previous with *pat*
$	last	*x-n*	*n* before *x*
+	next	*x,y*	*x* through *y*
-	previous	´*x*	marked with *x*
+*n*	*n* forward	´´	previous context

%	1,$

Initializing Options

EXINIT	place set's here in environment variable
$HOME/.exrc	editor initialization file
./.exrc	editor initialization file
set *x*	enable option
set no*x*	disable option
set *x*=*val*	give value *val*
set	show changed options
set all	show all options
set *x*?	show value of option *x*

Most Useful Options

autoindent	ai	supply indent
autowrite	aw	write before changing files
ignorecase	ic	in scanning
lisp	lisp	( ) { } are *s-exp*'s
list	list	print CONTROL-I for tab, $ at end
magic	magic	. [* special in patterns
number	nu	number lines
paragraphs	para	macro names which start . . .
redraw	redraw	simulate smart terminal
scroll	scroll	command mode lines
sections	sect	macro names
shiftwidth	sw	for < >, and input CONTROL-d
showmatch	sm	to) and } as typed
showmode	smd	show insert mode in vi
slowopen	slow	stop updates during insert
window	window	visual mode lines
wrapscan	ws	around end of buffer?
wrapmargin	wm	automatic line splitting

Scanning Pattern Formation

^	beginning of line
$	end of line
.	any character
\<	beginning of word
\>	end of word
[*str*]	any char in *str*
[↑*str*]	. . . not in *str*
[*x*-*y*]	. . . between *x* and *y*
*	any number of preceding

EXAMPLES

The command:

```
ex text
```

would invoke the editor with the file named `text`.

LIMITATIONS

The undo (`u`) command causes all marks to be lost on lines changed and then restored if the marked lines were changed.

The undo command never clears the "buffer modified" condition; that is, once the editor buffer has been modified, `ex` tells you that it is `[Modified]`, even if you undo the only modification.

The `z` command prints a number of logical rather than physical lines. More than a screen full of output may result if long lines are present.

File input/output errors don't print a name if the command line - option is used.

There is no easy way to do a single scan ignoring case.

The editor does not warn if text is placed in named buffers and not used before exiting the editor.

Null characters are discarded in input files, and cannot appear in resultant files.

FILES

`/usr/bin/e`
Executable file

`/usr/bin/ex`
Executable file

`/usr/bin/edit`
Executable file

`awk`(1), `ed`(1), `grep`(1), `sed`(1), `vi`(1)

`curses`(3X), `term`(4), `terminfo`(4) in *A/UX Programmer's Reference*

"Using `ex`" in *A/UX Text Editing Tools*

"Using `vi`" in *A/UX Text Editing Tools*

NAME

expand, unexpand — expand tabs to spaces, and vice versa

SYNOPSIS

expand -a [*-tabstop*] [*-tab1 , tab2 , ... , tabn*] [*file*]...

unexpand [*file*]...

ARGUMENTS

-a Inserts tabs whenever they would compress the resultant file by replacing two or more characters.

file Specifies the file to be read and expanded.

-tabstop
Sets the tabs at *tabstop* spaces apart instead of the default of 8, if a single tabstop is given.

-tab1 , tab2 , ... , tabn
Sets the tabs at the specified columns.

DESCRIPTION

expand reads the named files (or the standard input, if none are given) and writes on the the standard output with tabs changed into blanks. Backspace characters are preserved into the output and decrement the column count for tab calculations. expand is useful for preprocessing character files that contain tabs (before sorting, looking at specific columns, and so forth).

unexpand puts tabs back into the data from the standard input or the named files and writes the result on the standard output. By default, only leading blanks and tabs are reconverted to maximal strings of tabs.

FILES

/usr/ucb/expand
Executable file

/usr/ucb/unexpand
Executable file

See diction(1)

NAME

expr — evaluates arguments as an expression

SYNOPSIS

expr *arguments*

ARGUMENTS

arguments
Specifies the arguments to be evaluated.

DESCRIPTION

expr evaluates its arguments as an expression and writes the result on the standard output. Terms of the expression must be separated by blanks. Characters special to the Bourne shell or Korn shell (sh(1) or ksh(1), respectively) must be escaped. (The expr command is replaced in the C shell (csh(1)) by @.) Note that 0 is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be enclosed in quotation marks. Integer-valued arguments may be preceded by a unary minus sign. Internally, integers are treated as 32-bit, 2's-complement numbers.

The operators and keywords are listed below. Characters that need to be escaped are preceded by \. The list is in order of increasing precedence, with equal precedence operators grouped within { } symbols.

expr \| *expr*
returns the first *expr* if it is neither null nor 0; otherwise, returns the second *expr*.

expr \& *expr*
returns the first *expr* if neither *expr* is null or 0; otherwise, returns 0.

expr { =, \>, \>=, \<, \<=, != } *expr*
returns the result of an integer comparison if both arguments are integers; otherwise, returns the result of a lexical comparison.

expr { +, - } *expr*
addition or subtraction of integer-valued arguments.

expr { *, /, % } *expr*
multiplication, division, or remainder of the integer-valued arguments.

expr : *expr*
matching operator : compares the first argument with the second argument which must be a regular expression; regular expression syntax is the same as that of ed(1), except that all patterns are *anchored* (that is, begin with ^) and, therefore, ^ is not a special character, in that context. Normally, the matching operator returns the number of characters matched (0 on failure). Alternatively, the \(. \) pattern symbols can be used to return a portion of the first argument.

EXAMPLES

```
a='expr $a + 1'
```

adds 1 to the shell variable `a`.

```
# 'For $a equal to either "/usr/abc/file"
# or just "file"'
expr $a : '.*/\(.*\)'  \ | $a
```

returns the last segment of a pathname (that is, *file*). Watch out for / alone as an argument: *expr* will take it as the division operator (see the ''Limitations'' section later in this manual page).

A better representation of the preceding example:

```
expr //$a : '.*/\(.*\)'
```

the addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

```
expr $VAR  :  '.*'
```

returns the number of characters in `$VAR`.

STATUS MESSAGES AND VALUES

`syntax error`
for operator/operand errors

`non-numeric argument`
if arithmetic is attempted on such a string

As a side effect of expression evaluation, `expr` returns the following exit values:

`0` if the expression is neither null nor 0

`1` if the expression is null or 0

`2` for invalid expressions

LIMITATIONS

After argument processing by the shell, `expr` cannot tell the difference between an operator and an operand except by the value. If `$a` is an =, the command:

```
expr  $a  =  '='
```

looks like:

```
expr  =  =  =
```

as the arguments are passed to `expr` (and they will all be taken as the = operator). The following works:

```
expr  X$a  =  X=
```

FILES

`/bin/expr`
Executable file

SEE ALSO

csh(1), ed(1), ksh(1), sh(1)

NAME

f77 — invokes the Fortran 77 compiler

SYNOPSIS

f77 [-1] [-66] [-A *factor*] [-c] [-C] [-E] [-f] [-F] [-g] [-I[*24s*]] [-m] [-N*table*entries]... [-o*output*] [-O] [-onetrip] [-p] [-R] [-S] [-u] [-U] [-w] *file...*

ARGUMENTS

-1 Acts the same as the -onetrip option.

-66
Compiles as a Fortran 66 program.

-A *factor*
Expands the default symbol table allocations for the assembler and link editor. The default allocation is multiplied by the factor given.

-c Suppresses link editing and produces .o files for each source file.

-C Generates code for run-time subscript range-checking.

-E Indicates that the remaining characters in the argument are used as an EFL flag argument whenever processing a .e file.

-f Uses a version of f77 that handles floating-point constants and links the object program with the floating-point interpreter, in systems without floating-point hardware.

file Specifies the file to be processed through the Fortran 77 compiler.

-F Applies EFL preprocessor to relevant files and puts the result in files whose names have their suffix changed to .of. (No .o files are created.)

-g Generates additional information needed for the use of sdb(1)

-I[*24s*]
Changes the default size of integer variables (only valid on machines where the normal integer size is not equal to the size of a single precision real). -I2 causes all integers to be 2-byte quantities, -I4 (default) causes all integers to be 4-byte quantities, and -Is changes the default size of subscript expressions (only) from the size of an integer to 2 bytes.

-m Applies the M4 preprocessor to each EFL source file before transforming with the efl(1) processor.

-N*table*entries
Sets the maximum number of table entries to the number *entries*. Replaces *table* with one of the following letter designations corresponding to a compiler table:

q Uses the equivalence table.

x Uses the external names table.

s Uses the statement number table.

c Uses the control block table.

n Uses the identifier table.

To allow up to 1000 statement numbers, use `-Ns1000` as the option and argument.

`-o`*output*
Names the final output file *output*, instead of `a.out`.

`-O` Invokes an object code optimizer.

`-onetrip`
Performs all DO loops at least once. (Fortran 77 DO loops are not performed at all if the upper limit is smaller than the lower limit.)

`-p` Prepares object files for profiling (see `prof`(1)).

`-R` Removes the dynamically created assembler input file upon completion.

`-S` Compiles the named programs and leaves the assembler language output in corresponding files whose names are suffixed with `.s`. (No `.o` files are created.)

`-u` Makes the default type of a variable *undefined*, rather than using the default Fortran rules.

`-U` Does not ''fold'' cases. By default, the `f77` compiler is not case sensitive. This option causes `f77` to treat upper and lower cases separately.

`-w` Suppresses all warning messages. If the option is `-w66`, only Fortran 66 compatibility warnings are suppressed.

DESCRIPTION

`f77` is the Fortran 77 compiler; it accepts several types of *file* arguments:

Arguments whose names end with `.f` are taken to be Fortran 77 source programs; they are compiled and each object program is left in the current directory in a file whose name is that of the source, with `.o` substituted for `.f`. However, if a single Fortran source program is compiled and loaded all at one time, the `.o` file is deleted. By default the process produces an executable file, named `a.out`, in the current directory.

Arguments whose names end with `.r` or `.e` are taken to be EFL source programs; these are first transformed by the EFL preprocessor, then compiled by `f77`, producing `.o` files.

In the same way, arguments whose names end with `.c` or `.s` are taken to be C or assembly source programs and are compiled or assembled, producing `.o` files.

The options have the same meaning as in `cc` (see `ld`(1) for link editor options).

Other arguments are taken to be link editor option arguments, `f77`-compatible object programs (typically produced by an earlier run), or libraries of `f77`-compatible routines. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the default name `a.out`.

STATUS MESSAGES AND VALUES

The messages produced by `f77` itself are self-explanatory. The link editor, `ld`(1), may occasionally write messages upon the output stream(s).

FILES

`/usr/bin/f77`
: Executable file

`file.[fresc]`
: Input file

`file.o`
: Object file

`a.out`
: Linked output

`./fort` *[pid]*`.?`
: Temporary file

`/usr/lib/f77comp`
: Compiler file

`/lib/c2`
: Optional optimizer file

`/usr/lib/libF77.a`
: Intrinsic function library file

`/usr/lib/libI77.a`
: Fortran I/O library file

`/lib/libc.a`
: C library file

SEE ALSO

`asa`(1), `cc`(1), `efl`(1), `fpr`(1), `fsplit`(1), `ld`(1), `m4`(1), `prof`(1), `sdb`(1)

"`f77` Command Syntax," in *A/UX Programming Languages and Tools, Volume 1*

NAME

`factor` — prints the prime factor of a given number

SYNOPSIS

`factor` [*number*]

ARGUMENTS

number

Specifies the number from which `factor` will print its prime factors.

DESCRIPTION

`factor` prints the prime factors of its argument. When `factor` is invoked without an argument, it waits for a number to be typed in. If you type in a positive number less than `pow`(2,56), it will factor the number and print its prime factors; each one is printed the proper number of times. Then it waits for another number. It exits if it encounters a zero or any non-numeric character.

If `factor` is invoked with an argument, it factors the number as above and then exits.

Maximum time to factor is proportional to $\sqrt{n}$ and occurs when n is prime or the square of a prime, where n is the *number* being factored. It takes 1 minute to factor a prime near 10^{11} on a 68020.

STATUS MESSAGES AND VALUES

The message:

```
Ouch
```

is echoed when input is out of range or is garbage.

FILES

`/bin/factor`

Executable file

See true(1)

NAME

fcnvt — converts a file in one storage format to a different storage format

SYNOPSIS

fcnvt [-f] [-v] [-i *start-format*] -s *input-file output-file*

fcnvt [-f] [-v] [-i *start-format*] -d *input-file output-file*

fcnvt [-f] [-v] [-i *start-format*] -t *input-file output-file*

fcnvt [-f] [-v] [-i *start-format*] -p *input-file output-file*

fcnvt [-f] [-v] [-i *start-format*] -b *input-file output-file*

fcnvt [-f] [-v] [-i *start-format*] -m *input-file output-file*

ARGUMENTS

-b Converts the input file to BinHex 4.0 format. The input file is encoded into ASCII characters, permitting ASCII transfer of a binary file.

-d Converts the input file to Apple Double format.

-f Allows fcnvt to overwrite an existing file with the same name as the new file.

-i *start-format*
Specifies the current format of the file to be converted. If an input-file format is not specified in this way, the Apple Single format is assumed.

input-file
Specifies the file to be converted.

-m Converts the input file to a MacBinary format. This format is commonly used to transfer Macintosh files by means of a telecommunications link, using a protocol such as XMODEM, XMODEM7, Kermit, CompuServe A, or CompuServe B.

output-file
Specifies the name of the file to be created in the desired storage format.

-p Converts the input file to Plain Pair format. This option is the same as the -t option except that *output-file*.info is not created.

-s Converts the input file to Apple Single format. This format is the default.

start-format
Specifies the present format of the file to be converted, in terms of one of the following formats:

single

```
double
triple
pair
bin
hex
```

These values correspond to the Apple Single, Apple Double, Plain Triple, Plain Pair, BinHex 4.0, and MacBinary formats as explained in ''File Formats'' in the ''Description'' section.

`-t` Converts the input file to Plain Triple format. Three files are created; suffixes are used to distinguish the three derived files. (See ''File Formats'' in the ''Description'' section.)

`-v` Specifies verbose mode. In verbose mode, `fcvnt` displays information as it processes each file.

DESCRIPTION

`fcnvt` converts a file in one storage format to a corresponding file in a different storage format.

The Finder application fully supports only the Apple Double and Apple Single formats; that is, you can open files in either of these two formats by double-clicking or by choosing Open from the File menu of the Finder.

The main purpose of `fcnvt` is to convert Macintosh files received in a foreign format to one of the native formats of Apple Single or Apple Double.

File formats other than the genuine Apple Double and Apple Single formats are needed because the native file formats permit files to contain data values that make electronic transmission unreliable. Even with the best communications programs, files containing all the possible binary data values can cause transmission problems. Therefore, a conversion process is the only really safe method for transmitting files.

The Macintosh Operating System takes advantage of a twin-file storage scheme. Whenever applicable, certain types of data are stored in a resource-fork file as well as a data-fork file. For users, this fact is normally irrelevant because the Finder represents these twin files with a single icon, and both files are manipulated simultaneously as if they were one file.

File Formats

Of the file formats that can be obtained, the most desirable ending format is a Finder-compatible one. The Finder can decipher and open files in Apple Single and Apple Double file formats only.

With Apple Single (`single`) format, data and file-attribute information is kept in a single file. Apple Single format is best used for nontext data and executable Macintosh object files. Directory listings look much cleaner

because each Macintosh file is mapped to a single A/UX file with no percent sign (%) prefix. A header at the beginning of the file contains offsets to the data, resource information, or both, as well as information corresponding to Macintosh file attributes.

With Apple Double (`double`) format, the contents of the data fork are stored in one file, known as the data file; resources and file-attribute information are stored in a separate file, known as the header file. The header file has the same name as the data file, but the filename is prefixed with a percent sign. The Apple Double format is best used for text data and data to be shared with UNIX utilities, because the data fork is available as an isolated file.

The Plain Triple (`triple`) format is used by the `macget` and `macput` public-domain file-transfer programs. The files *output-file*`.info`, *output-file*`.data`, and *output-file*`.rsrc` contain identification information, the data fork, and the resource fork, respectively.

The Plain Pair (`pair`) format is similar to the `triple` format except that *output-file*`.info` is not created.

The BinHex 40 format converts all binary and any other data into an ASCII-encoded form, permitting an ASCII transfer mode for what were once binary files.

The MacBinary storage format is commonly used to transfer Macintosh files over a telecommunications link, using a protocol that supports certain kinds of binary file transfers, such as XMODEM, XMODEM7, Kermit, CompuServe A, or CompuServe B.

File Transfers

Among UNIX users, files are often distributed electronically, and often they are compressed as well as converted into easily transmittable formats. To recover the transmitted file to a form that is usable, the expansion and conversion steps must be performed in the reverse order in which these steps were performed before transmission.

To illustrate the simpler case (no file compression before transmission), you could make the file `/FileMakerII` ready for transmission by converting it to a MacBinary format with

```
fcnvt -m /FileMakerII /tmp/filemaker
```

Once received by the receiving party, the file can be converted back to an Apple Single format with this command:

```
fcnvt -i bin filemaker.bin FileMakerII
```

A MacBinary format is commonly used to send Macintosh files over a telecommunications link using a protocol such as XMODEM, XMODEM7, Kermit, CompuServe A, or CompuServe B.

A file downloaded onto your system through a terminal emulator program is likely to be in text-only format, BinHex 4.0 format, or MacBinary format, or it may simply be a copy of the resource fork of the Macintosh file. The preferred way to discover the format is to receive that information directly from the sender.

Note that file transfers made through terminal emulators are likely to strip away the Macintosh type and creator attributes for the file. (Each of these attributes is one four-character string.) See `setfile`(1) for information on restoring these attributes. If they are missing, the preferred way to discover these attributes is to receive that information directly from the sender.

FILES

`/mac/bin/fcnvt`
Executable file

SEE ALSO

`setfile`(1), `tar`(1), `uuencode`(1C), `uusend`(1C)

See grep(1)

NAME

file — determines the type of a file

SYNOPSIS

file [-c] [-f *ffile*] [-m *mfile*] *arg*...

ARGUMENTS

arg Specifies the argument that will be tested by the file command.

-c Causes file to check the magic file for format errors. This validation is not normally carried out for reasons of efficiency. No file typing is done under the -c option.

-f *ffile*
Specifies the file containing the names of the files to be examined.

-m *mfile*
Instructs file to use an alternate magic file.

DESCRIPTION

file performs a series of tests on each file named in its arguments in an attempt to classify it. If the file appears to be ASCII, file examines the first 512 bytes and tries to guess its language. If a file is an executable a.out file, file will print the version stamp, provided it is greater than 0 (see ld(1)).

The file command uses the file /etc/magic to identify files that have some sort of ''magic number,'' that is, any file containing a numeric or string constant that indicates its type. Commentary at the beginning of /etc/magic explains its format.

EXAMPLES

The command:

file *text-file program-file directory*

reports the filenames and directory name, and whether the files are English text, nroff input, a C program, or whatever.

FILES

/bin/file
Executable file

/etc/magic
Executable file

SEE ALSO

ld(1)

magic(4) in *A/UX Programmer's Reference*

NAME

find — finds files

SYNOPSIS

find *pathname... expression*

ARGUMENTS

expression
Specifies the Boolean expression that is to be found.

pathname
Specifies the pathname to be searched.

DESCRIPTION

find recursively descends the directory hierarchy for each pathname specified by a *pathname* argument, seeking files that match the Boolean expression *expression* written in the primaries described in this section. The find command does not follow symbolic links. In the descriptions of the primaries, the argument *n* represents a decimal integer, where +*n* means more than *n*, -*n* means less than *n*, and *n* means exactly *n*.

-atime *n*
True if the file has been accessed within the last *n* days. The access time of directories in the *pathname* argument is changed by find itself.

-cpio *device*
Always true; writes the current file on *device* in cpio(4) format (512-byte records).

-ctime *n*
True if the file has been changed within the last *n* days.

-depth
Always true; causes descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. This argument can be useful when you use find with cpio(1) to transfer files that are contained in directories for which you do not have write permission.

-exec *cmd*
True if the executed command *cmd* returns a zero value as its exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument of the form {} is replaced by the current pathname.

(*expression*)
True if the parenthesized expression is true. (Parentheses are special to the shell and must be escaped.)

`-fstype`*type*
True if the file system to which the file belongs is of the type specified by *type*. The value of *type* can be `4.2` Berkeley Software Distribution (BSD), `5.2` System V File System (SVFS), or `nfs` Network File System (NFS).

`-group` *gname*
True if the file belongs to the group *gname*. If *gname* is numeric and does not appear in the `/etc/group` file, it is interpreted as a group ID.

`-inum` *n*
True if the file has the inode number specified by *n*.

`-links` *n*
True if the file has *n* links.

`-mtime` *n*
True if the file has been modified within the last *n* days.

`-name` *file*
True if *file* matches the current filename. You can use normal shell argument syntax if it is escaped. (Use the `[`, `?`, and `*` characters carefully.)

`-newer` *file*
True if the current file has been modified more recently than the argument *file*.

`-ok` *cmd*
Same as `-exec`, except that the generated command line is printed with a question mark as the first character, and is executed only if you respond by typing `y`.

`-perm` *onum*
True if the file permission flags exactly match the octal number *onum* (see `chmod`(1)). If *onum* is prefixed by a minus sign, more flag bits (017777, described in `stat`(2)) become significant and the flags are compared:

```
(flags & onum) == onum
```

`-print`
Always true; causes the current pathname to be printed.

`-prune`
Always true. Has the side effect of pruning the search tree at the file; that is, if the current pathname is a directory, `find` does not descend into that directory.

`-size` *n*[c]
: True if the file is *n* blocks long (512 bytes per block). If *n* is followed by a `c`, the size is in characters.

`-type` *c*
: True if the type of the file is *c*, where *c* is `b`, `c`, `d`, `l`, `p`, or `f` for block special file, character special file, directory, symbolic link, FIFO (named pipe), or plain file, respectively.

`-user` *uname*
: True if the file belongs to the user *uname*. If *uname* is numeric and does not appear as a login name in the `/etc/passwd` file, it is interpreted as a user ID.

You can combine the primaries by using the following operators (in order of decreasing precedence):

1. The negation of a primary. The exclamation point (!) is the unary NOT operator.
2. Concatenation of primaries. The AND operation is implied by the juxtaposition of two primaries.
3. Alternation of primaries. `-o` is the OR operator.

EXAMPLES

The following command locates and prints all files in `/tmp` and its subdirectories that are named `junk`.

```
find /tmp -name junk -print
```

The following command finds all files, starting with the root directory, on which the permission levels have been set to 755 (see `chmod`(1)).

```
find / -perm 755 -exec ls "{}" ";"
```

With `-exec` and a command such as `ls`, it is often necessary to escape the curly braces (`{}`), that store the current pathname under investigation, by putting the pathname in double quotation marks. It is *always* necessary to escape the semicolon at the end of an `-exec` sequence.

Note again that it is also necessary to escape parentheses used for grouping primaries, by means of a backslash, as shown here. The following command removes all files named `a.out` or `*.o`.

```
find \( -name a.out -o -name '*.o' \) -exec rm {} \;
```

FILES

`/bin/find`
: Executable file

`/etc/passwd`
: File containing user information

```
find /tmp -name junk -print
```

`/etc/group`
File containing group information

SEE ALSO

`chmod`(1), `cpio`(1), `csh`(1), `ksh`(1), `sh`(1), `xargs`(1)

`stat`(2), `cpio`(4), `fs`(4) in *A/UX Programmer's Reference*

`ff`(1M) in *A/UX System Administrator's Reference*

"Other Programming Tools" in *A/UX Programming Languages and Tools, Volume 2*

NAME

finger — displays information about the users on a system

SYNOPSIS

finger [f] [w] [*login-or-real-name*]...

finger -i [f] [w] [*login-name*]...

finger -q [f] [w] [*login-name*]...

finger -l [b] [h] [m] [p] [*login-or-real-name*]...

finger [-l] *login-or-real-name*@*host* [*login-or-real-name*@*host*]...

finger [-s] @*host* [@*host*]...

ARGUMENTS

@*host*

Specifies the name of a remote host. The value of *host* can be a host name or an Internet address, as specified in the /etc/hosts file. If you specify just a remote host, finger ignores any options that you may have specified and displays the unmodified default format for each user on the remote host. If you precede *host* with a login or a real name, finger ignores any options that you may have specified and displays the unmodified long format for that user.

-b Suppresses the display of the Directory and Shell fields when the long format is used.

-f Suppresses the display of column headings.

-h Suppresses the display of the first line of the .project file.

-i Causes finger to show the Login, TTY, When, and Idle columns for only those users who have idle time. If you specify *login-name*, it is the login name of a user, as stored in the first field of /etc/passwd, because finger searches only that field for a match.

-l Causes finger to use the long format. You should use this option only in conjunction with options that modify the long format.

login-name

Specifies the name of the user for which you want information.

login-or-real-name

Specifies the login name or any of the real names stored in the fifth field of the user's entry in /etc/passwd. For example, if Christine Louise Witt's login name is chris, and Tina Louise Witt is in the fifth field of her entry in /etc/passwd, finger can display information about her if you use any of the following names as a value of the *login-or-real-name* argument: chris, Tina, Louise, or

Witt. Note that if you specify Christine as the value of the *login-or-real-name* argument in this case, finger cannot find a match and displays this message:

```
Login name: christine          In real life: ???
```

-m Limits the search for a match to the login name, as stored in the first field of /etc/passwd.

-p Suppresses the display of any .plan files when the long format is used.

-q Causes finger to display only the Login, TTY, and When columns. If you specify *login-name*, it is the login name of a user, as stored in the first field of /etc/passwd, because finger searches only that field for a match.

-s Causes finger to use the default format. You should use this option only in conjunction with options that modify the default format.

-w Suppresses display of the Name column.

DESCRIPTION

The finger command displays information about the users who are currently logged in to the system. On A/UX systems, finger displays a line of information for each user logged in to the system and for each CommandShell window.

The finger command displays information about the users on a system in either of two formats: the default format and the long format. The default format is displayed when you do not specify any options or arguments; when you specify the -i, -q, or -s option; or when you specify the name of a remote system. The long format is displayed when you specify the -l option or when you specify the name of a user on the local system or on a remote system.

Default Format

The default format produces fields of data that have these headings: Login, Name, TTY, Idle, When, and Office. The Login column is the user's login name as stored in the first field of the /etc/passwd file. The Name column is the real name of the user as stored in the fifth field of the /etc/passwd file. The TTY column is the user's current terminal name. The Idle column is the amount of time since the user last pressed a key on the keyboard. When the value in the TTY column is co, the value in the When column represents the time the user logged in to the system. For the lines that correspond to a CommandShell window, the value in the When column represents the time the user created the window. The Office column reflects extra information, if any, taken from the fifth field of the user's entry in /etc/passwd. See "Enhancing finger

Output'' later in the ''Description'' section for details.

Status of permission to write to the user's terminal is indicated by the presence (denied) or absence (enabled) of an asterisk (*) before the terminal name. If idle time is a single integer, it represents minutes. If idle time is two integers separated by a colon (:), they represent hours and minutes. If idle time is an integer to which a d is appended, it represents days.

The -f, -i, -l, -q, and -w options modify the default format.

Long Format

When you use the long format, finger displays the same information as would be displayed in the default format, as well as the user's home directory and shell, the contents of the user's .plan file, if any, and the first line of the user's .project file, if any.

The long format displays this information in the following format:

```
Login name: login-name          In real life: real-names
Mailstop: mail-stop             Home phone: phone-number
Directory: directory            Shell: shell
On since date-and-time on ttytime Idle Time
Project: project-information
Plan: plan-information
```

The -b, -h, @*host*, -m, login-or-real-name, -p, and -s options modify the long format.

Enhancing finger Output

You can use the chfn command to store extra information in the fifth field of your entry in /etc/passwd. The extra information consists of your office number or mail stop, office phone number, and home phone number. If this information is available, finger displays the office number or mail stop and office phone number when the default format is used and displays the office number or mail stop, office phone number, and home phone number when the long format is used.

FILES

/usr/ucb/finger
: Executable file

/etc/passwd
: File that is examined for login names, real names, and other information

/etc/utmp
: File that finger uses to determine who is currently logged in

/usr/adm/lastlog
: File that contains the last login time of each user

`~/.plan`
File containing the user's plan

`~/.project`
File describing the projects the user is working on

SEE ALSO

`chfn`(1), `w`(1), `who`(1), `whoami`(1)

`hosts`(4), `passwd`(4) in *A/UX Programmer's Reference*

`in.fingerd`(1M) in *A/UX System Administrator's Reference*

NAME

fmt — invokes a simple text formatter

SYNOPSIS

fmt [*file*]...

ARGUMENTS

file Specifies the file to be formatted.

DESCRIPTION

fmt is a simple text formatter which reads the concatenation of input files (or standard input if none are given) and produces on standard output a version of its input with lines as close as possible to 72 characters long. The spacing at the beginning of the input lines is preserved in the output, as are blank lines and interword spacing.

The fmt command is meant to format mail messages prior to sending, but may also be useful for other simple tasks. For instance, within visual mode of the ex editor (e.g. vi) the command:

```
!}fmt
```

will reformat a paragraph, evening the lines.

LIMITATIONS

The program was designed to be simple and fast; for more complex operations, the standard text processors are likely to be more appropriate.

FILES

/usr/ucb/fmt
Executable file

SEE ALSO

nroff(1), mail(1), pr(1), troff(1), vi(1)

NAME

fold — folds long lines for finite-width output device

SYNOPSIS

fold [*-width*] [*file*]...

ARGUMENTS

file Specifies the file containing the lines that will be folded.

-width

Specifies the maximum width for the lines. The default is 80.

DESCRIPTION

fold is a filter which will fold the contents of the specified files, or the standard input if no files are specified, breaking the lines to have maximum width. The width should be a multiple of 8 if tabs are present, or the tabs should be expanded using expand before coming to fold.

LIMITATIONS

If underlining is present it may be corrupted by folding.

FILES

/usr/ucb/fold

Executable file

SEE ALSO

expand(1)

NAME

fpr — filters the output of Fortran programs for line printing

SYNOPSIS

fpr

DESCRIPTION

fpr is a filter that transforms files formatted according to Fortran's carriage control conventions into files formatted according to UNIX line printer conventions.

The fpr command copies its input onto its output, replacing the carriage control characters with characters that will produce the intended effects when printed using lpr(1). The first character of each line determines the vertical spacing as follows:

Character	Vertical Space Before Printing
Blank	One line
0	Two lines
1	To first line of next page
+	No advance

A blank line is treated as if its first character is a blank. A ''blank'' that appears as a carriage control character is deleted. A ''0'' is changed to a newline. A ''1'' is changed to a form feed. The effects of a ''+'' are simulated using backspaces.

EXAMPLES

The following lines show two ways to use this command:

```
a.out | fpr | lpr

fpr < f77.output | lpr
```

LIMITATIONS

Results are undefined for input lines longer than 170 characters.

FILES

/usr/ucb/fpr
Executable file

SEE ALSO

asa(1), f77(1)

NAME

freq — reports character frequencies in a file

SYNOPSIS

freq [*file*]...

ARGUMENTS

file Specifies the file that will be reported on. If no file is specified, the standard input is read.

DESCRIPTION

freq counts occurrences of characters in the list of files specified on the command line.

EXAMPLES

The command:

```
freq filea
```

will list a count of each character that appears in filea.

FILES

/bin/freq
Executable file

NAME

from — displays the mail header lines in your mailbox

SYNOPSIS

from [-s *sender*] [*user*]

ARGUMENTS

-s *sender*
: Prints only the headers for mail that is sent by *sender*.

user
: Causes from to examine the *user*'s mailbox instead of your own.

DESCRIPTION

from prints out the mail header lines in your mailbox file to show you who your mail is from.

The from command works with mail and mailx.

FILES

/usr/ucb/from
: Executable file

/usr/bin/mailx
: Executable file

/usr/mail/*
: Mail files

SEE ALSO

biff(1), mail(1), mailx(1)

NAME

fsplit — splits f77 or efl files

SYNOPSIS

fsplit [-e] [-f] [-s] *file*...

ARGUMENTS

-e Indicates that the input files are EFL.

-f Indicates that the input files are F77.

file Specifies the file to be split.

-s Strips f77 input lines to 72 or fewer characters with trailing blanks removed.

DESCRIPTION

fsplit splits the named *file*s into separate files, with one procedure per file. A procedure includes blockdata, function, main, program, and subroutine program segments. Procedure *X* is put in file *X*.f, *X*.r, or *X*.e depending on the language flag option chosen, with the following exceptions: main is put in the file MAIN.[efr] and unnamed blockdata segments in the files blockdata*N*.[efr] where *N* is a unique integer value for each file.

FILES

/bin/fsplit
Executable file

SEE ALSO

csplit(1), efl(1), f77(1), split(1)

NAME

fstyp — reports the file-system type

SYNOPSIS

fstyp *file*

ARGUMENTS

file Specifies the file residing on the file system that will be reported.

DESCRIPTION

fstyp reports the type of the file system on which *file* resides. If *file* is a device file, fstyp attempts to read a file-system superblock from the device. The file system must be one of the supported types listed in fstypes.

STATUS MESSAGES AND VALUES

If successful, fstyp prints to the standard output a message that indicates the file system type.

FILES

/etc/fstyp
Executable file

/etc/fs/*/fstyp
Executable file

SEE ALSO

statfs(2), fstyp(3), fs(4), fstypes(4) in *A/UX Programmer's Reference*

NAME

ftp — transfers files by using the DARPA Internet File Transfer Protocol (FTP)

SYNOPSIS

ftp [-d] [-g] [-i] [-n] [-v] [*remote-system*]

ARGUMENTS

-d Enables debugging. By default, debugging is disabled. You can also enable and disable debugging by running the debug command from the ftp command interpreter.

-g Disables filename expansion. You can also enable and disable filename expansion by running the glob command from the ftp command interpreter.

-i Enables or disables interactive prompting during multiple file transfers. You can also enable and disable prompting by running the prompt command from the ftp command interpreter.

-n Disables the automatic login process, called "auto login," that ftp normally performs upon initial connection. By default, auto login is enabled. See "The Autologin Process" later in the "Description" section for details. If you use this option, you can log in after connecting to the remote system by running the user command from the ftp command interpreter.

remote-system
Specifies the name of the remote system with which ftp is to connect. The value of *remote-system* can be the host name or the Internet address of a system that is reachable through an Ethernet connection. If you provide this argument, ftp immediately attempts to establish a connection to an FTP server on the specified system. If you do not provide this argument, ftp enters its command interpreter and waits for your commands.

-v Causes ftp to display all responses from the remote system and report data-transfer statistics. You can also enable and disable this display by running the verbose command from the ftp command interpreter.

DESCRIPTION

ftp allows you to transfer files to and from a remote system by using the DARPA Internet File Transfer Protocol. To use ftp, your system must be running a kernel that includes the Berkeley networking software. Your interface to ftp is through its command interpreter.

The ftp Command Interpreter

To indicate that it is waiting for you to enter a command, ftp displays this prompt:

```
ftp>
```

This section describes the commands that ftp recognizes. To preserve embedded spaces in command arguments, you can enclose the command in double quotation marks ("). Many of the ftp commands, such as bell and prompt, are "toggle switches" in the sense that running the command enables or disables the command, depending on the command's previous state. When you run one of these commands, ftp reports the new command state.

! [*command* [*arg* ...]]
: Runs an interactive shell on the local system. If you specify a *command* argument, it is taken to be a command to execute, and any *arg* arguments are passed as arguments to *command*.

$ *macro-name* [*arg* ...]
: Runs the macro *macro-name*, as defined by the macdef command, described later in this list, or as defined in your .netrc file, described in "The .netrc File" later in the "Description" section.

  Arguments specified by *arg* arguments are not expanded before being passed to the macro. The macro processor interprets the dollar sign ($) and backslash (\) as special characters. A $ followed by one or more numbers is replaced by the corresponding argument on the macro invocation command line. A $ followed by an i tells the macro processor that the executing macro is to be looped. On the first pass, $i is replaced by the first argument on the macro invocation command line; on the second pass, it is replaced by the second argument; and so on. A \ followed by any character is replaced by that character. Use \ to prevent special treatment of the $.

? [*command*]
: Acts as a synonym for the help command.

account [*passwd*]
: Specifies a supplemental password, which may be required by a remote system for access to resources once you are successfully logged in. If you do not provide a *passwd* argument, ftp prompts for the account password and disables echoing as you enter the password.

append *local-file* [*remote-file*]
: Appends a local file to a file on the remote system. If *remote-file* is not specified, the local filename is used in naming the remote file, after being altered by any ntrans or nmap setting. The file transfer is subject to the current settings for type, form, mode, and struct.

`ascii`
Sets the file-transfer type to network ASCII, which is the default.

`bell`
Enables or disables the sounding of a bell when each file transfer command completes.

`binary`
Sets the file transfer type to support binary image transfer.

`bye`
Disconnects from the remote server and exits `ftp`. Entering the end-of-file character (usually CONTROL-D) has the same effect.

`case`
Enables or disables case mapping during `mget` commands. When `case` is enabled, those filenames on the remote system whose letters are all uppercase are written on the local system with the letters mapped to lowercase. By default, `case` is disabled.

`cd` *remote-directory*
Changes the working directory on the remote system to the directory specified by *remote-directory*.

`cdup`
Changes the working directory on the remote system to the parent of the current working directory on the remote system.

`close`
Disconnects from the remote server and returns to the command interpreter. Any defined macros are erased.

`cr` Enables or disables the stripping of carriage return characters during the retrieval of ASCII files. By default, `cr` is disabled.

When the value of `type` is `ascii`, `ftp` interprets carriage return-newline sequences as record delimiters. When `cr` is enabled, `ftp` strips carriage returns from this sequence to conform with the UNIX single-newline record delimiter.

Records on non-UNIX remote systems may contain single newlines. When the value of `type` is `ascii`, `ftp` can distinguish these newlines from a record delimiter only when `cr` is disabled.

`debug` [*debug-value*]
Enables or disables debugging mode. By default, `debug` is disabled.

The optional argument *debug-value* is an integer value that specifies a debugging level. The default value is 1; higher values cause `ftp` to display more detailed debugging information. When debugging is on, `ftp` displays each command sent to the remote system, preceded by

this string:

```
-->
```

`delete` *remote-file*
Deletes the file specified by *remote-file* on the remote system.

`dir` [*remote-directory*] [*local-file*]
Displays a list of the contents of the directory specified by *remote-directory*. If a *local-file* argument is specified, the output is placed in the file specified by *local-file*. If you do not specify a *remote-directory* argument, the contents of the current working directory on the remote system are displayed. If you do not specify a *local-file* argument, or if the value of *local-file* is a hyphen (`-`), the output is displayed on the terminal.

`disconnect`
Acts as a synonym for the `close` command.

`form` *format*
Sets the file-transfer form to *format*. The default format is `file`.

`get` *remote-file* [*local-file*]
Retrieves the remote file specified by *remote-file* and stores it on the local system. If you do not specify a *local-file* argument, the transferred file is given the same name that it has on the remote system, subject to alteration by the current `case`, `ntrans`, and `nmap` settings. The current settings for `type`, `form`, `mode`, and `struct` affect the file transfer.

`glob`
Enables or disables filename expansion for the `mdelete`, `mget`, and `mput` commands. When `glob` is disabled, the filename arguments are taken literally and not expanded. When `glob` is enabled, filename expansion for `mput` is done as in the C shell. For `mdelete` and `mget`, each remote filename is expanded separately on the remote system and the lists are not merged. Expansion of a directory name is likely to be different from expansion of the name of an ordinary file because the exact result depends on the foreign operating system and the FTP server. You can preview the filename expansion that will result by running this command:

`mls` *filename-argument ...* `-`

`hash`
Enables or disables the display of a number sign (`#`) for each data block transferred. The size of a data block is 1024 bytes. By default, `hash` is disabled.

`help` [*command*]
Prints an informative message about the meaning of the command specified by *command*. If you do not specify a *command* argument, `ftp` displays a list of the `ftp` commands.

`lcd` [*directory*]
Changes the working directory on the local system. If you do not specify a *directory* argument, the working directory is changed to your home directory.

`ls` [*remote-directory*] [*local-file*]
Displays an abbreviated list of the contents of a directory on the remote system. If you do not specify a *remote-directory* argument, `ftp` displays the contents of the current working directory. If you do not specify a *local-file* argument, or if *local-file* is a hyphen (`-`), the output is displayed on the terminal.

`macdef` *macro-name*
Defines a macro and causes `ftp` to enter macro input mode. Lines that you subsequently enter are stored as the macro named by *macro-name* until you enter a null line to terminate macro input mode. You can use a total of at most 4096 characters to define at most 16 macros. Macros remain defined until you issue the `close` command.

`mdelete` [*remote-file* ...]
Deletes the files on the remote system specified by one or more *remote-file* arguments.

`mdir` *remote-file* ... *local-file*
Displays a list of the contents of the directories specified by one or more *remote-file* arguments. This command is similar to the `dir` command except that you can specify multiple *remote-file* arguments. If the *local-file* argument is specified, the output is placed in the file specified by *local-file*. If `prompt` is enabled, `ftp` prompts you to verify that the last argument is indeed the target local file for receiving `mdir` output. If the value of *local-file* is a hyphen (`-`), the output is displayed on the terminal.

`mget` *remote-file* ...
Expands the files on the remote system specified by one or more *remote-file* arguments and does a `get` for each filename thus produced. See `glob` for details on filename expansion. Resulting filenames are then processed according to `case`, `ntrans`, and `nmap` settings. Files are transferred into the local working directory.

The `mget` command is not meant to transfer entire directory subtrees of files. You can transfer a subtree by transferring a `tar` archive of the subtree in binary mode.

`mkdir` *directory-name*
Makes the directory specified by *directory-name* on the remote system.

`mls` *remote-file ... local-file*
Lists information about the files specified by one or more *remote-file* arguments and places the output in the file specified by *local-file*. If *local-file* is a hyphen (`-`), the output is displayed on the terminal. If `prompt` is enabled and *local-file* is not a hyphen, `ftp` prompts you to verify that the last argument is indeed the target local file for receiving `mls` output.

`mode` [*mode-name*]
Sets the file-transfer *mode* to the value specified by *mode-name*. The only supported mode is stream mode.

`mput` *local-file ...*
Expands the list of local files specified as *local-file* arguments and does a `put` operation for each file in the resulting list. See the description of `glob` earlier in this list, for details of filename expansion. Resulting filenames are then processed according to `ntrans` and `nmap` settings.

The `mput` command is not meant to transfer entire directory subtrees of files. You can transfer a subtree by transferring a `tar` archive of the subtree in binary mode.

`nmap` [*inpattern outpattern*]
Sets or unsets the filename-mapping mechanism. If you do not specify any arguments, the filename-mapping mechanism is unset. If you specify arguments, remote filenames are mapped during execution of `mput` and `put` commands issued without a specified remote target filename, and local filenames are mapped during execution of `mget` commands and `get` commands issued without a specified local target filename.

This command is useful when connecting to a non-UNIX remote system that has different file-naming conventions or practices. The mapping follows the pattern set by *inpattern* and *outpattern*. The value of *inpattern* is a template for incoming filenames, which may have already been processed according to the `ntrans` and `case` settings. Variable templating is accomplished by including the sequences `$1`, `$2`,...,`$9` in *inpattern*. Use `\` to prevent this special treatment of the `$` character. All other characters are treated literally and are used to determine the `nmap` *inpattern* variable values.

The *outpattern* argument determines the resulting mapped filename. The sequences `$1`, `$2`,...,`$9` are replaced by any value resulting

from the *inpattern* template. The sequence `$0` is replaced by the original filename. Additionally, the sequence [*seq1*, *seq2*] is replaced by *seq1* if *seq1* is not a null string; otherwise, it is replaced by *seq2*.

For example, given an *inpattern* argument whose value is `$1.$2` and the transfer of a remote file named `mydata.data`, `$1` has the value `mydata`, and `$2` has the value `data`. Given an *outpattern* argument whose value is

```
[$1,$2].[$2,file]
```

the output filename is `myfile.data` for input filenames `myfile.data` and `myfile.data.old`; `myfile.file` for the input filename `myfile`; and `myfile.myfile` for the input filename `.myfile`.

You can include space characters in *outpattern*, as in this example:

```
nmap $1 | sed "s/  *$//" > $1
```

Use the `\` character to prevent special treatment of the `$`, `[`, `]`, and `,` characters.

`ntrans` [*inchars* [*outchars*]]
Sets or unsets the filename-character-translation mechanism. If you do not specify any arguments, the filename-character-translation mechanism is unset. If you specify *inchars* and *outchars* arguments, characters in remote filenames are translated during execution of `mput` and `put` commands issued without a specified remote target filename, and characters in local filenames are translated during execution of `mget` and `get` commands issued without a specified local target filename.

This command is useful when you are connecting to a non-UNIX remote system that has different file-naming conventions or practices. Characters in a filename matching a character in *inchars* are replaced with the corresponding character in *outchars*. If the character's position in *inchars* is longer than the length of *outchars*, the character is deleted from the filename.

`open` *remote-system* [*port*]
Establishes a connection to the FTP server running on the system specified by *remote-system*. If you supply an optional port number, `ftp` attempts to contact an FTP server at that port. If you do not disable auto login by invoking `ftp` with the `-n` option, `ftp` automatically attempts to log the user in to the FTP server. See ''The Autologin Process'' later in the ''Description'' section for details.

`prompt`
Enables or disables interactive prompting. By default, `prompt` is

enabled. Interactive prompting occurs during multiple file-transfer operations operations to allow you to send or receive files selectively. If prompting is disabled, any `mget` or `mput` transfers all files, and any `mdelete` deletes all files.

`proxy` *ftp-command*
Executes an `ftp` command on a secondary control connection. This command allows simultaneous connection to two remote FTP servers for transferring files between the two servers. The first `proxy` command should be an `open` command to establish the secondary control connection. Enter the command `proxy ?` to see other `ftp` commands available on the secondary connection. The following commands behave differently when prefaced by `proxy`:

`open`
Does not define new macros during the auto-login process.

`close`
Does not erase existing macro definitions.

`get`
`mget`
Transfer files from the host on the primary control connection to the host on the secondary control connection.

`put`
`mput`
`append`
Transfer files from the host on the secondary control connection to the host on the primary control connection. Third-party file transfers depend upon support of the File Transfer Protocol `PASV` command by the server on the secondary control connection.

`put` *local-file* [*remote-file*]
Sends a local file to the remote system. If you do not specify a *remote-file* argument, the local filename is used to name the remote file, after processing according to any `ntrans` or `nmap` settings. The file transfer is subject to the current settings for `type`, `form`, `mode`, and `struct`.

`pwd`
Displays the name of the current working directory on the remote system.

`quit`
Acts as a synonym for the `bye` command.

`quote` *arg ...*
Sends the arguments, specified by one or more *arg* arguments, to the

remote FTP server.

`recv` *remote-file* [*local-file*]
Acts as a synonym for the `get` command.

`remotehelp` [*command-name*]
Displays a list of commands for which the remote FTP server can provide help. If you specify a *command-name* argument, the server responds with the help information for the specified command.

`rename` [*current-name*] [*new-name*]
Renames the file specified by *current-name* on the remote system to the name specified by *new-name*.

`reset`
Clears the reply queue. This command resynchronizes command and reply sequencing with the remote FTP server. Resynchronization may be necessary following a violation of the File Transfer Protocol by the remote server.

`rmdir` *directory-name*
Deletes a directory on the remote system.

`runique`
Enables or disables the use of unique names to store files on the local system. By default, `runique` is disabled.

When `runique` is enabled and a file already exists with a name equal to the target local filename for a `get` or `mget` command, the characters `.1` are appended to the name. If that name matches another existing file, the characters `.2` are appended to the original name, and so on, until `ftp` generates a unique name, in which case it displays the name on the terminal. If this process continues through `.99`, `ftp` displays an error message and the transfer does not take place. Note that `runique` does not affect local files generated from a shell command.

`send` *local-file* [*remote-file*]
Acts as a synonym for the `put` command.

`sendport`
Enables or disables the use of PORT commands. By default, `ftp` attempts to use a PORT command when establishing a connection for each data transfer. The use of PORT commands can prevent delays when you are performing multiple file-transfer operations. If the PORT command fails, `ftp` uses the default data port. When PORT commands are disabled, `ftp` makes no attempt to use PORT commands for each data transfer. This function is useful for certain FTP implementations that ignore PORT commands but indicate,

incorrectly, that they have been accepted.

`status`
Shows the current status of the connection, including command settings.

`struct` [*struct-name*]
Sets the file-transfer structure to the value specified by *struct-name*. The only supported structure is the ''file'' structure.

`sunique`
Enables or disables the use of unique filenames when sending files to a remote system. By default, `sunique` is disabled. The remote FTP server must support the File Transfer Protocol STOU command to enable this command successfully. See `runique` for details.

`tenex`
Sets the file-transfer type to that needed for TENEX systems.

`trace`
Enables or disables packet tracing. By default, `trace` is disabled.

`type` [*type-name*]
Sets the file-transfer type to the value specified by *type-name*. The value of *type-name* can be `ascii`, `binary`, `ebcdic`, `image`, or `tenex`. The values `binary` and `image` are equivalent.

If you do not specify a *type-name* argument, `ftp` displays the current type. By default, `type` is ASCII.

`user` *user-name* [*password*] [*account*]
Logs you in to the remote FTP server. If you do not specify a *password* argument and the server requires a password, `ftp` prompts for a password after disabling local echo. If you do not specify *account* and the FTP server requires an account name, `ftp` prompts for it. If you specify an *account* argument and the remote server does not require an *account* argument, `ftp` relays an ACCT command to the remote server after the login sequence is completed. Unless you disable auto login by using the `-n` option, this process occurs automatically when the connection to the FTP server is first established.

`verbose`
Enables or disables verbose mode. By default, `verbose` is enabled. In verbose mode, `ftp` displays all responses from the FTP server and displays file-transfer statistics. Note that when `verbose` is disabled, `ftp` does not display any responses from the FTP server, including the output for such commands as `pwd`.

Stopping a File Transfer

To stop sending a file transfer, send an interrupt, which is usually done by pressing CONTROL-C. Sending transfers are halted immediately. You can halt receiving transfers by sending a File Transfer Protocol ABOR command to the remote server and discarding any further data received. The speed at which this function is accomplished depends upon the remote server's support for ABOR processing. If the remote server does not support the ABOR command, an `ftp>` prompt does not appear until the remote server has finished sending the requested file.

Interrupts are ignored when `ftp` has completed any local processing and is waiting for a reply from the remote server. A long delay while waiting may result from the ABOR processing just described or from unexpected behavior by the remote server, including violations of the File Transfer Protocol. If the delay results from unexpected remote server behavior, you can stop the local instance of `ftp` only by running the `kill` command.

File-Naming Conventions

Files specified as arguments to `ftp` commands are processed according to the following rules:

1. If the filename is a hyphen (`-`), the standard input and standard output are used for reading and writing, respectively.
2. If the first character of the filename is an exclamation mark (`!`), `ftp` interprets the remainder of the argument as a shell command. To run the command, `ftp` forks a shell, passes any specified arguments, and reads or writes from the standard output or standard input as necessary. If the shell command includes spaces, the argument must be quoted. Here is an example:

   ```
   "!ls -lt"
   ```

3. If the filename is not a hyphen and does not begin with an exclamation mark, and if filename expansion is enabled, local filenames are expanded according to the rules used in the C shell. See the `glob` command for details. If you use an `ftp` command that expects a single local file, such as `put`, only the first filename generated by filename expansion is used.
4. For `mget` and `get` commands with unspecified local filenames, the local filename is the remote filename, which may be altered by a `case`, `ntrans`, or `nmap` setting. The resulting filename may then be altered if `runique` is enabled.
5. For `mput` and `put` commands with unspecified remote filenames, the remote filename is the local filename, which may be altered by a `ntrans` or `nmap` setting. The resulting filename may then be altered by the remote server if `sunique` is enabled.

File-Transfer Parameters

These parameters affect file transfer: `type`, `mode`, `form`, and `struct`. See the description of `type` in the "Description" section for its possible settings. This version of `ftp` supports only the default values for `mode`, `form`, and `struct`.

The Autologin Process

By default, when you use the *remote-system* argument on the `ftp` command line to specify the name of a remote system to connect to, `ftp` automatically prompts you for the login name that you want to use on the remote system. If, when prompted, you press RETURN, `ftp` uses your login name on the local system. Next, `ftp` prompts for a password. Some FTP servers also prompt for a second password. This sequence of prompts and responses is known as the auto-login process.

If you have a `.netrc` file in your home directory and it contains an entry for the system specified by *remote-system*, `ftp` extends the auto-login process by using the entry to log you in automatically. See "The `.netrc` File" later in the "Description" section for details.

Whether you have a `netrc` file or not, you can disable the auto-login process by running `ftp` with the `-n` option. In this case, use the `open` command to log in to the FTP server.

The `.netrc` File

The `.netrc` file contains login and initialization information used by the auto-login process. The file resides in your home directory and must be readable only by you. An entry in this file consists of these tokens, separated by a space, a tab, or a newline character:

`account` *string*
: Specifies an additional account password. If this token is present, the auto-login process supplies the specified string if the remote server requires an additional account password, or the auto-login process initiates an `account` command if it does not. Not all FTP servers support this command.

`login` *login-name*
: Specifies a login name on the remote system. If this token is present, the auto-login process initiates a login session using the specified name.

`macdef` *name*
: Defines a macro. This token provides the same functionality as the `ftp macdef` command and causes the creation of a macro whose name is *name*. The macro definition begins with the next line in `.netrc` and continues until the auto-login process reads a null line. If you define a macro named `init`, it is automatically executed as the

last step in the auto-login process.

`machine` *remote-system*
Specifies the name of a remote system. The auto-login process searches the `.netrc` file for a `machine` token that matches the *remote-system* argument specified on the `ftp` command line or as an argument to the `open` command. Once a match is made, the system processes the subsequent `.netrc` tokens, stopping when another `machine` token is encountered or the end of the file is reached.

`password` *string*
Specifies a password. If this token is present, the auto-login process supplies the specified string if the remote server requires a password as part of the login process. Note that if this token is present in the `.netrc` file, `ftp` stops the auto-login process if the `.netrc` file is readable by anyone other than you.

Anonymous FTP

The FTP server prohibits logging in to an account that does not have a password. To make files available to anyone who can connect to the system, some system administrators set up the FTP server to allow logging in as `anonymous` or `ftp`, which are special login names for which any password is acceptable. If you log in anonymously, you are restricted to the home directory of the user `ftp`. See `ftpd`(1M) for details.

EXAMPLES

The first example illustrates a simple `ftp` connection and file transfer from the remote system to the local system. Long output lines have been folded for the sake of readability.

```
ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
Password:
230 User tim logged in.
ftp> get tmac.an
200 PORT command successful.
150 Opening data connection for tmac.an
                    (89.0.0.33,1205) (13366 bytes).
226 Transfer complete.
local: tmac.an remote: tmac.an
13922 bytes received in 0.69 seconds (20 Kbytes/s)
ftp> quit
221 Goodbye.
```

The second example illustrates an `ftp` connection and a file transfer from the local system to the remote system.

```
ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
Password:
230 User tim logged in.
ftp> put tmac.an
200 PORT command successful.
150 Opening data connection for tmac.an
                     (89.0.0.33,1209).
226 Transfer complete.
local: tmac.an remote: tmac.an
13922 bytes sent in 0.83 seconds (16 Kbytes/s)
ftp> quit
221 Goodbye.
```

The third example illustrates changing to a directory in the remote file system and listing the contents of several directories.

```
ftp printms
Connected to printms.
220 printms FTP server (Version 4.109 Fri Nov 20
                    07:43:57 PST 1987) ready.
Name (printms:tim):
331 Password required for tim.
Password:
230 User tim logged in.
ftp> ls
200 PORT command successful.
150 Opening data connection for /bin/ls
                          (89.0.0.33,1212) (0 bytes).
OUT
cutmks
tmac.ap
tmac.ptx
tmac.syn
tmac.toc
226 Transfer complete.
76 bytes received in 1.2 seconds (0.13 Kbytes/s)
ftp> cd OUT
250 CWD command successful.
```

```
ftp> ls
200 PORT command successful.
150 Opening data connection for /bin/ls
                    (89.0.0.33,1213) (0 bytes).
junk
226 Transfer complete.
4 bytes received in 0.058 seconds (3.9 Kbytes/s)
ftp> close
221 Goodbye.
ftp> quit
```

LIMITATIONS

Correct execution of many commands depends upon proper behavior by the remote server.

An error in the treatment of returns in the 4.2 BSD UNIX ASCII-mode transfer code has been corrected. This correction may result in incorrect transfers of binary files to and from 4.2 BSD servers using the `ascii` type. You can avoid this problem by using the `image` or `binary` type.

When verbose mode is disabled, `ftp` does not echo responses from the remote server or any responses to the `pwd` command.

The FTP server disconnects any login sessions that have had no activity for a certain amount of time. The default period is 15 minutes.

FILES

`/usr/bin/ftp`
: Executable file

`/usr/spool/ftp/*`
: File hierarchy to which, by convention, anonymous FTP users are restricted

SEE ALSO

`csh`(1), `tar`(1)

`ftpd`(1M) in *A/UX System Administrator's Reference*

A/UX Networking Essentials

A/UX Command Reference was written, edited, and composed on a desktop publishing system using Apple Macintosh computers, and `troff` running on A/UX. Page proofs were created on Apple LaserWriter printers. Final pages were output directly to 70 millimeter film on an Electrocomp 2000 Electron Beam Recorder. PostScript, the page-description language for the LaserWriter, was developed by Adobe Systems Incorporated.

Text type and display type are Times, Garamond, and Helvetica. Bullets are ITC Zapf Dingbats®. Some elements, such as program listings, are set in Apple Courier, a fixed-width font.

Writers: Erik Akin, Mike Elola, Kristi Fredrickson, Michael Hinkson, Linda Kinnier, Paul Pannish, Cheryl Salgado, Kathy Wallace, and Laura Wirth

Writing Group Lead: Mike Elola

Developmental Editor: Silvio Orsino

Art Director: Tamara Whiteside

Production Editor: Jeannette Allen

Production Supervisor: Robin Kerns

Special thanks to Anne Szabla and Chris Wozniak