AT&T

System V

Interface Definition

2

System V

Interface Definition

System V Interface Definition

System V
Interface Definition

Issue 2

Volume II

ISBN 0-932764-10-X

Library of Congress Catalog Card No. 85-063224

Select Code No. 320-012

IMPORTANT NOTE TO USERS

While every effort has been made to ensure the accuracy of all information in this document, **AT&T** assumes no liability to any party for any loss or damage caused by errors or omissions or by statements of any kind in the *System V Interface Definition*, its updates, supplements, or special editions, whether such errors are omissions or statements resulting from negligence, accident, or any other cause. **AT&T** further assumes no liability arising out of the application or use of any product or system described herein; nor any liability for incidental or consequential damages arising from the use of this document. **AT&T** disclaims all warranties regarding the information contained herein, whether expressed, implied or statutory, including implied warranties of merchantability or fitness for a particular purpose.

AT&T makes no representation that the interconnection of products in the manner described herein will not infringe on existing or future patent rights, nor do the descriptions contained herein imply the granting of license to make, use or sell equipment constructed in accordance with this description.

AT&T reserves the right to make changes without further notice to any products herein to improve reliability, function, or design.

This document was set on an **AUTOLOGIC, Inc. APS-5** phototypesetter driven by the `troff` formatter operating on **UNIX** System V on an **AT&T 3B20** computer.

---

* **UNIX** is a trademark of **AT&T**.
APS-5 is a trademark of **AUTOLOGIC, Inc.**.

How to Order

To order copies of the *System V Interface Definition* by phone, you may call:

(800) 432-6600 (Inside U.S.A.)
(800) 255-1242 (Inside Canada)
(317) 352-8557 (Outside U.S.A. & Canada)

You must use a major credit card for orders made by phone.

To order copies of the *System V Interface Definition* by mail, write to:

AT&T Customer Information Center (CIC)
Attn: Customer Service Representative
P.O. Box 19901
Indianapolis, IN 46219
U.S.A.

Be sure to include the address the books should be shipped to and a check or money order made payable to **AT&T**.

Please identify the books you want to order by Select Code. Select Codes for the *System V Interface Definition* are:

320-011	Volume I
320-012	Volume II
307-127	All Volumes

Table of Contents

Preface

The *System V Interface Definition* specifies an operating system environment that allows users to create applications software that is independent of any particular computer hardware. The *System V Interface Definition* applies to computers that range from personal computers to mainframes. Applications that conform to this specification will allow users to take advantage of changes in technology and to choose the computer that best meets their needs from among many manufacturers while retaining a common computing environment.

The *System V Interface Definition* specifies the operating system components available to both end-users and application programs. The functionality of components is defined, but the implementation is not. The *System V Interface Definition* specifies the source-code interfaces of each operating system component as well as the run-time behavior seen by an application program or an end-user. The emphasis is on defining a common computing environment for application programs and end-users; not on the internals of the operating system, such as the scheduler or memory manager.

An application program using only components defined in the *System V Interface Definition* will be compatible with and portable to any computer that supports the System V Interface. While the source-code may have to be re-compiled to move an application program to a new computer system that supports the System V Interface, the presence and behavior of the operating system components as defined by the *System V Interface Definition* would be assured.

The *System V Interface Definition* is organized into a Base System Definition plus a series of Extension Definitions. The Base System Definition specifies the components that all System V operating systems must provide. The Extensions to the Base System are not required to be present in a System V operating system, but when a component is present it must conform to the specified functionality. The *System V Interface Definition* lets end-users and application developers identify the features and functions available to them on any System V operating system.

Part I

A General Introduction to the System V Interface Definition

Chapter 1
General Introduction

1.1 AUDIENCE AND PURPOSE

The *System V Interface Definition* (SVID) is intended for use by anyone who must understand the operating system components that are consistent across all System V environments.

As such, its primary audience is the application developer building C language application programs whose source-code must be portable from one System V environment to another. A system builder should also view these volumes as a necessary condition for supporting a System V environment that will host such applications.

This publication is intended to serve the following major purposes:

- To serve as a single reference source for the definition of the external interfaces to services that are provided by all System V environments. These services are designated as the Base System. This includes source-code interfaces and run-time behavior as seen by an application program. It does not include the details of how the operating system implements these functions.
- To define all additional services (such as networking and data management) at an equivalent external interface level and to group these services into Extensions to the Base System.
- To serve as a complete definition of System V external interfaces, so that application source-code that conforms to these interfaces and is compiled in an environment that conforms to these interfaces, will execute as defined in a System V environment. It is assumed that source-code is recompiled for the proper target hardware. The basic objective is to facilitate the writing of application program source-code that is directly portable across all System V implementations. Facilities outside of the Base System would require that the appropriate Extension be installed on the target environment.

1.2 STRUCTURE AND CONTENT

1.2.1 Partitioning into Base System and Extensions

The *System V Interface Definition* partitions System V components into a Base System and Extensions to that Base System. This does not change the definition of System V. It is instead a recognition that some of the functionality of System V may be unnecessary in certain environments, especially on small hardware configurations. It also recognizes that different computing environments require some functions that others do not.

The Base System functionality has been structured to provide a minimal, stand-alone run-time environment for applications originally written in a high-level language, such as C. In this environment, the end-user is not expected to interact directly with the traditional System V shell and commands. An example of such a system would be a dedicated-use system. That is, a system devoted to a single application, such as a vertically integrated application package for managing a legal office, To execute, many applications programs will require only the components in the Base System. Other applications will need one or more Extensions.

The Extensions to this Base System have been structured to provide a growth path in natural functional increments that leads to a full System V configuration. The division between Base and Extensions will allow system builders to create machines tailored for different purposes and markets, in an orderly fashion. Thus, a small business/professional computer system designed for novice single users might include only the Base System and the Basic Utilities Extension. A system for advanced business/professional users might add to this the Advanced Utilities Extension. A system designed for high-level language software development would include the Base System, the Kernel Extension and the Basic Utilities, Advanced Utilities, and Software Development Extensions. Although the Extensions are not meant to specify the physical packaging of System V for a particular product, it is expected that the Extensions will lead to a fairly consistent packaging scheme.

This partitioning allows an application to be built using a basic set of components that are consistent across all System V implementations. This basic set is the Base System. Where necessary, an application developer can choose to use components from an Extension and require the run-time environment to support that Extension in addition to the Base System.

Facilities or side effects that are not explicitly stated in the SVID are not guaranteed, and should not be used by applications that require portability.

1.2.2 Conforming Systems

All conforming systems must support the source code interfaces and runtime behavior of the components of the Base System. A system may conform to none or some Extensions. All the components of an Extension must be present for a system to meet the requirements of the Extension. This does not preclude a system from including only a few components from some Extension, but the system would *not* then be said to have the Extension. Some Extensions require that other Extensions be present on a system, for example, the Advanced Utilities Extension requires the Basic Utilities Extension.

This issue of the *System V Interface Definition* corresponds to functionality in AT&T System V Release 1.0 and System V Release 2.0. An implementation of System V may conform to the System V Release 1.0 functionality or the System V Release 2.0 functionality. All System V Release 2.0 enhancements to System V Release 1.0 are identified as such in the SVID.

1.2.3 Organization of Technical Information

For ease of use, the SVID has been divided into several Volumes containing the following Extensions:

Volume 1. Base System

Kernel Extension

Volume 2. Basic Utilities Extension

Advanced Utilities Extension

Software Development Extension

Administered System Extension

Terminal Interface Extension

Additional Volumes will define any further Extensions to System V.

The SVID defines the source-code interface and the run-time behavior of the components that make up the Base System and each Extension. Components include, for example, operating system service routines, general library routines, system data files, special device files, and end-user utilities (commands).

When referred to individually, components will be identified by a suffix of the form (XX_YYY) where XX identifies the Base System or the Extension that the component is in and YYY identifies the type of the component. For example, components defined in the Operating System Service Routines section of the Base System will be identified by (BA_OS), components defined in General Library Routines of the Base System will be identified by (BA_LIB), and components defined in the Operating System Service Routines section of the Kernel Extension will be identified by (KE_OS). Possible types are OS, LIB, CMD (commands or utilities) and ENV (environment).

The definition of the Base System includes an overview followed by chapters that provide detailed definitions of each component in the Base System. Similarly, the definition of each Extension includes an overview followed by chapters that provide detailed definitions of each component in the Extension.

Pages containing the detailed component definitions are labeled with the name of the component being defined. Some utilities and routines are described with other related utilities or routines, and therefore do not have detailed definition pages of their own.

An alphabetical index is provided in each Volume listing all components defined in that Volume. The index points to the detailed definition pages on which a component is to be found; the header for these pages may not contain the name of the component being sought. For example, in Volume I, the entry for the function `calloc` points to the MALLOC(BA_OS) pages, because the function `calloc` is defined with the function `malloc` on pages labeled MALLOC(BA_OS).

Each component definition follows the same structure. The sections are listed below; not all the following sections may be present in each description. If present, however, they will be in the given order. Sections entitled **EXAMPLE, APPLICATION USAGE**, and **USAGE** are not considered part of the formal definition of a component.

- **NAME** — name of component
- **SYNOPSIS** — summary of source-code or user-level interface
- **DESCRIPTION** — interface and runtime behavior
- **RETURN VALUE** — value returned by the function
- **ERRORS** — possible error condidions
- **FILES** — names of files used
- **APPLICATION USAGE** or **USAGE** — guidance on use
- **EXAMPLE** — example
- **SEE ALSO** — list of related components
- **FUTURE DIRECTIONS** — planned enhancements
- **LEVEL** — see MECHANISM FOR EVOLUTION below

In general, components that are utilities do not have a **RETURN VALUE** section. Except as noted in the detailed definition for a particular utility, utilities return a zero exit code for *success*, and non-zero for *failure*.

The component definitions are similar in format to AT&T System V manual pages, but have been extended or modified as follows:

- All machine-specific or implementation-specific information has been removed. All implementation-specific constants have been replaced by symbolic names, which are defined in a separate section [see **Implementation-specific constants** in **Volume I: Part II — Base System Definition: Chapter 4 — Definitions**]. When these symbolic names are used they always appear in curly brackets, e.g., {PROC_MAX}. The symbolic names correspond to those defined by the November 1985 draft of the **IEEE P1003** Standard to be in a `<limits.h>` header file; however, in this document, they are *not* meant to be read as symbolic constants defined in header files.
- A section entitled **FUTURE DIRECTIONS** has been added to selected component definitions. This section indicates how a component will evolve. The information ranges from specific changes in functionality to more general indications of proposed development.
- A section entitled **APPLICATION USAGE** or **USAGE** has been added to guide application developers on the expected or recommended usage of certain components. Detailed definitions of operating system services and library routines have an **APPLICATION USAGE** paragraph while utilities have a **USAGE** paragraph. While operating system services and library routines are only used by

programs, utilities may be used by programs, by end-users or by system administrators. The **USAGE** paragraph indicates which of these three is appropriate for a particular utility (this is not meant to be prescriptive, but rather to give guidance). The following terms are used in the **USAGE** paragraph: *application program*, *end-user*, *system administrator*, or *general*. The term *general* indicates that the utility might be used by all three: application programs, end-users and system administrators.

- A section entitled **LEVEL** defines each component's commitment level:

  Level-1 components will remain in the SVID and can be modified only in upwardly compatible ways. Any change in its definition will preserve the previous source-code interface and run-time behavior in order to ensure that the component remains upwardly compatible.

  Level-2 components will remain unchanged for at least three years following entry into level-2, after which time the component may be modified in a non-upwardly compatible way or may be dropped from the SVID. Level-2 components are labeled with the starting date of this three-year period.

1.3 MECHANISM FOR EVOLUTION

The SVID will be reissued as necessary to reflect developments in the System V Interface. In conjunction with these updates, the following changes may be made to the definitions:

- Level-1 components may be moved to Level-2. The date of their entry into Level-2 will be the date of the reissue of the SVID in which the change is made.
- Level-1 components will *not* move from one Extension into another Extension.
- Components may move *from* existing Extensions *into* the Base System. Components will *not* move *from* the Base System *into* an Extension.
- New Extensions may be introduced with completely new functionality.

1.4 C LANGUAGE DEFINITION

Source-code interfaces described in the SVID are for the C language.

The following two references define the C language for System V Release 1.0 and System V Release 2.0 respectively:

- *UNIX™ System V Programming Guide*, Issue 1, February 1982.
- *UNIX™ System V Programming Guide*, Issue 2, April 1984.

Chapter 2
Future Directions

2.1 NETWORK SERVICES EXTENSION

The Network Services Extension will provide advanced standard interfaces to support networking applications. It is divided into three functional areas. The Open Systems Networking Interfaces section describes a protocol-independent application interface to transport services based on the Open Systems Interconnection (OSI) Reference Model [IS 7498]. The Streams I/O Interfaces section describes the operating system service routines that provide direct access to protocol modules implemented using the streams framework. The Shared Resource Environment section describes new capabilities for sharing and administering resources among interconnected machines.

2.2 OPERATING SYSTEM STANDARDS

The IEEE P1003 working group is currently pursuing a draft standard for a portable operating system interface. The *System V Interface Definition* is consistent with the trial-use standard (November 1985), with several minor exceptions. Full conformance to the IEEE standard will be strongly considered after its formal approval.

2.3 C LANGUAGE STANDARDIZATION

AT&T is committed to support the standardization of the C language being pursued by ANSI X3J11, in which its representatives take a leading role. Full conformance to the ANSI standard will be strongly considered after formal approval.

2.4 FLOATING POINT STANDARDS

The IEEE P754 Standard for Binary Floating Point Arithmetic will be supported by System V. The existing library routines that deal with floating point numbers, and which are likely to change in order to support the IEEE P754 Standard, belong to the following classes:

- routines that do arithmetic operations;
- routines that do input/output;
- routines that manipulate floating point numbers.

However, these changes are hardware dependent and will appear only on the machines whose underlying floating point data representation and exception handling mechanisms are those specified by the IEEE P754 Standard.

2.5 GRAPHICS EXTENSION

This Extension will track current industry efforts to define standards for graphics functions. One area under active consideration is the Graphical Kernel Subsystem (GKS).

2.6 TERMINAL INTERFACE EXTENSION

The current Terminal Interface Extension consists of the facilities provided by the curses/terminfo package to allow application programs to perform terminal-handling functions in a way that is independent of the type of terminal acutally in use. This Extension will be enhanced to support applications on both character and bit-mapped terminals and to provide capabilities for handling windows, menus, icons, etc. which can be accessed by a keyboard or other input device, such as a mouse. Applications written in this environment will have a uniform and easily used human interface. In addition, applications which rely on curses/terminfo will be compatible with the new environment.

2.7 INTERNATIONALIZATION

Where necessary, modifications will be made, in an upwardly compatible way, to existing System V components to support internationalization. In addition, new components will be added to support features not currently available in System V. These will include tools that will allow *national supplements* to be added to an implementation of System V.

National supplements would be small packages that contained the necessary supplementary information, such as messages, databases, documentation, and device-drivers that, when installed, would allow an implementation of System V to process different national languages and support hardware (i.e., terminals, printers) and local conventions found in different countries. System builders would be able to create national supplements using the tools provided in System V.

More than one national supplement could be installed on a system at a time, resulting in a system with multiple language capabilities; however, national supplements are envisioned as self-contained, not requiring or depending on other installed national supplements.

Facilities that System V will provide to support internationalization and the development of national supplements are:

- Messages and text from the kernel, utilities, and application programs will be separated to enable support for national languages.
- Local conventions, or *environments*, will be supported transparently, depending on the language selected by the user. Among the conventions that will be supported are date and time formats, collating sequences, and numeric representations.
- Supplementary code-sets will be supported. This will allow use of multiple code-sets, and consequently character symbols, in addition to the ASCII code-set.

- Sixteen-bit code-sets will be supported. This will allow languages of Far Eastern countries (Japan, Republic of China, Korea, the People's Republic of China, etc.) to be used.
- Language selection will be provided at the user-level to allow users of different languages to use the same system at the same time in their respective languages.

Message Handling. In the future, System V will support a facility to produce messages and text in national languages. In conjunction with the *Error Handling Standards* defined in **Volume I: Part II – Base System Definition: Chapter 7 – General Library Routines**, messages and text from the kernel, utilities, and applications would be stored separately. In addition, a set of administrative utilities would be provided to allow the creation of new messages and strings, as well as modification to existing ones.

Local Conventions. Local conventions define the common forms and rules used to communicate information. The aim of internationalization is to provide System V applications and utilities with the capability to interact with the end-user according to these local conventions. At the same time, applications and utilities must be portable and easily adapted to other conventions (i.e., they must be shielded from any particular set of conventions). Existing utilities and interfaces will be modified to support both implicit and explicit invocation of these conventions, with the following areas targeted for support:

Collating Sequence: The capability to define one or more *collating sequences* for a specific code-set will be provided. Utilities providing sorted output or requiring sorted input will be modified to allow invocation of different collating sequences. In addition, tools will be provided to support the definition of specific collating sequences.

Character Classification: The capability to define, on a language-by-language basis, character classes will be provided. The CTYPE(BA_LIB) library will be enhanced to provide character classification in local languages. Where possible, this capability will be provided through the existing classification routines. In addition, new routines will be provided to support new capabilities (i.e., returning an indication of which code-set a particular character comes from).

Date and Time Format: The capability to enter and display date and time in the local language and according to local formats will be provided. This applies to all utilities or services that operate with date/time specifications.

Numeric Representation: The capability to define the rules for numeric editing (such as decimal delimiter) will be provided.

Currency Representation: The capability to specify rules and formats for editing local currency will be provided.

8th-bit Cleanup. To support code-sets in addition to ASCII, all 8-bits of a byte will be used for character encoding. For example, some existing routines or utilities reject characters with octal values greater than 177. Future releases will eliminate this and similar problems.

Code-Set and Character Support. There are essentially two representations that make up the code-set:

the *external code-set* and the *internal code-set.*

The external code-set are those code-sets generated by input/output devices (i.e., terminals, printers, etc.). The most notable example is the seven-bit ASCII[1] code-set produced by most terminals and printers connected to System V today.

The internal code-set is a transformation of the external code-set according to the rules presented in this section, and is used to represent bytes throughout the rest of System V. Normally, no part of System V, except a device-driver, will see the external code-set; however, in many cases, the external and internal encodings will be the same with only minor exceptions.

The device-driver has the sole responsibility of mapping an external code-set to an internal code-set and vice-versa.

The following sections describe a template for transforming externally coded characters into internally coded characters, methods of designating a particular code-set to be used, and methods of designating a particular language to be used.

A *Code-Set Template* is a template for transforming externally coded characters into internally coded characters accessible by the System V operating system, utilities, and applications. The internal coding method discussed here is based on the **ISO 2022-1982** standard for code extension techniques, which suggests the following two techniques for shifting between code-sets:

- Single-shift
- Locking-shift

The single-shift is a single byte used to announce a temporary shift to another code-set. The byte, or bytes, immediately following the single-shift code are interpreted as part of a new code-set. Subsequent characters are interpreted as belonging to the primary code-set.

1. ASCII, as it is used here, is defined as the seven-bit code-set used for information interchange in the United States. It does *not* refer to the extended eight-bit ASCII code-set, sometimes known as ASCII-8, or local derivatives of the seven-bit ASCII code-set used in parts of Europe.

The ISO standard defines two single-shift characters:

1. SS2, or *single-shift two*, and
2. SS3, or *single-shift three.*

The SS2 character is represented by hexadecimal `8e`, while the SS3 character is represented by hexadecimal `8f`.

The locking-shift technique is used to temporarily *shift-in* and *shift-out* of code-sets. It consists of a pair of character sequences that allow a new code-set to be used for more than one character. While in the context of a locking-shift sequence, all characters, with the exception of single-shifted characters, are assumed to belong to the new code-set.

Because of the context sensitivity of the locking-shift sequence, this method is not recommended for use in System V. Therefore, the use of the single-shift sequence is recommended to reduce the context sensitivity to as little as possible.

In addition to using the single-shifts to distinguish characters, the eighth-bit will also be used to distinguish between the primary code-set and characters in one of the three supplementary code-sets. By using the combination of eighth-bit and single-shift characters, the internal coding method specifies a template for allowing four code-sets to coexist simultaneously: one primary code-set and three supplementary code-sets, with the two of the latter denoted by a single-shift character. The representations for these internal code-sets are shown below:

Code-Set	Internal Representation
Set 0 (Primary code-set)	`0XXXXXXX`

Set 1 (Supplementary code-set #1)	`1XXXXXXX`
	`– or –`
	`1XXXXXXX 1XXXXXXX`

Set 2 (Supplementary code-set #2)	`SS2 1XXXXXXX`
	`– or –`
	`SS2 1XXXXXXX 1XXXXXXX`

Set 3 (Supplementary code-set #3)	`SS3 1XXXXXXX`
	`– or –`
	`SS3 1XXXXXXX 1XXXXXXX`

Designation of the exact value of the four code-sets is performed through a code-set designation and is discussed in the following section.

A *Code-Set Designation* will be dynamic and accessible/modifiable at the operating system, utility and application levels to satisfy the specific needs for supporting multiple code-sets. It will also reside at the file level, so files with different code-set designations can exist on the same machine. That is, one file may be encoded with one set of code-sets while another file is encoded with another set of code-sets.

Specifically, it is desirable for code-set designation to meet the following requirements:

1. Code-set designations should be supported at the file level. Each file would contain its own set of code-set designation values.
2. At file creation time, all files would be designated with a system-wide default value.
3. Code-set designations could be changed dynamically.
4. The code-set designation value should contain information about:
   - The width of a character in the code-set,
   - The specific code-set designated (e.g., **DIS 8859/1**[2], **JIS 6226**[3], etc.),
5. Code-set designation information should be transferrable with the file contents across networks.

In addition to the code-set designation, a *language-designation* would offer the ability to designate which of several languages should be used for producing systems messages and for establishing an overall profile of the user's environment. One method under consideration for this type of designation is to use one or more exported environment variables. For example, a LANGUAGE variable would be used to denote the language (e.g., French, German, Italian, Japanese, English). This variable would also be used as an index to user profile information to determine which local conventions to use. The variable could be assigned at initiation of the login session and could also be changed at any time. In this way, language-designation is performed at user-level and controls the language of all system messages and text coming out of the operating system, utilities and applications, as well as particular national conventions.

Handling Non-standard Code-Sets. There are several code-sets in the world that the code-set template described here cannot support. The problem centers around the use of the eighth-bit to distinguish between characters in different code-sets. Specifically, these code-sets are as follows:

- The *shifted*-**JIS** code-set used in Japan,
- The packed Hangul code-set used in Korea,
- The **Big 5** code-set used in the Republic of China (Taiwan),
- The Chinese Code for Data Communcations also used in the Republic of China.

---

2. **DIS 8859/1** Latin Language no. 1 is the newly-adopted **ISO** standard code-set, supporting most of the Western European characters. It is an 8-bit code-set that contains **US ASCII** as a subset.
3. **JIS 6226** is a **ISO** standard code-set for supporting the Japanese language. It is a 16-bit code-set that contains both the hiragana and katakana alphabets, as well as about 7000 of the kanji ideograms.

Present plans are to provide limited support for these code-sets. Limited support means that files containing these code-sets could be stored on System V machines. No other suppport is currently planned; this implies that the mechanism for processing these files would have to be built into applications.

Character Support. In some applications it will be necessary to manipulate the variable-width characters coming from the supplementary code-sets. Although some application developers may choose to develop their own facilities for supporting this, System V will provide a generic facility for manipulating internally coded eight-bit bytes to a data type that can represent characters in a consistent manner. Initially, a new data type will be defined in the C programming language to support up to 16-bits of information. In addition, routines that use this new data type will be provided to allow application developers to perform operations on them.

Part II

Basic Utilities Extension Definition

Chapter 3
Introduction

3.1 OVERVIEW

The Basic Utilities Extension defines an environment that provides basic user-level functionality. It includes: the `sh` (shell) command interpreter, and shell programming aids; facilities for basic directory and file manipulation; and facilities for text file editing and processing.

The System V Base is a prerequisite for the Basic Utilities Extension.

The Advanced Utilities Extension provides the next logical expansion step up from the Basic Utilities Extension.

3.2 DESCRIPTION

UTILITIES

ar	diff	nl	sort
awk	dirname	nohup	spell
banner	du	pack	split
basename	echo	paste	sum
cal	ed	pcat	tail
calendar	expr	pg +	tee
cat	false	pr	test
cd	file	ps	touch
chmod	find	pwd	tr
cmp	grep	red	true
col	kill	rm	umask
comm	line	rmail	uname
cp	ln	rmdir	uniq
cpio	ls	rsh	unpack
cut	mail	sed	wait
date	mkdir	sh	wc
df	mv	sleep	

\+ New in System V Release 2.

Chapter 4
Commands and Utilities

NAME

ar — archive and library maintainer for portable archives

SYNOPSIS

```
ar option [ posname ] afile [name] ...
```

DESCRIPTION

The `ar` command maintains groups of files combined into a single archive file. It is used to create and update library files as used by the link editor [see LD(SD_CMD)]. It can be used, however, for any similar purpose. If an archive file is created from printable files, the entire archive file is printable.

Archives of text files created by `ar` are portable between implementations of System V.

When `ar` creates an archive file, it creates administrative information in a format that is portable across all machines. When there is at least one object file (that `ar` recognizes as such) in the archive, an archive symbol table is created in the archive file and maintained by `ar`. The archive symbol table is never mentioned or accessible to the user. (It is used by the link editor to search the archive file.) Whenever the `ar` command is used to create or update the contents of such an archive, the symbol table is rebuilt. The `s` modifier character described below forces the symbol table to be rebuilt.

The argument `option` is a `-` followed by one character from the set `drqtpmx` which may be optionally concatenated with one or more characters to modify the action. These modifier characters are taken from the set `vuabicls` but not all modifiers make sense with all options. See below for further explanation. The argument `posname` is the name of a file in the archive file, used for relative positioning; see options `-r` and `-m` below. The argument `afile` is the archive file. The `names` are constituent files in the archive file.

The meanings of the `option` characters are:

`-d` Delete the named files from the archive file. Valid modifiers are `vl`.

`-r` Replace the named files in the archive file. Valid modifiers are `vuabicl`. If the modifier `u` is used, then only those files with dates of modification later than the archive files are replaced. If an optional positioning character from the set `abi` is used, then the `posname` argument must be present, and specifies that new files are to be placed after (`a`) or before (`b` or `i`) `posname`. Otherwise new files are placed at the end.

`-q` Quickly append the named files to the end of the archive file. Valid modifiers are `vcl`. In this case `ar` does not check whether the added members are already in the archive. This is useful to bypass the searching otherwise done, when creating a large archive piece-by-piece.

`-t` Print a table of contents of the archive file. If no names are given, all files in the archive are listed. If names are given, only those files are

listed. Valid modifiers are `vs`. The `v` modifier gives a long listing of all information about the files.

`-p` Print the named files from the archive. Valid modifiers are `vs`.

`-m` Move the named files to the end of the archive. Valid modifiers are `vabil`. If a positioning modifier from the set `abi` is present, then the `posname` argument must be present, and, as with the option character `r`, it specifies where the files are to be moved.

`-x` Extract the named files. If no names are given, all files in the archive are extracted. The archive file is not changed. Valid modifiers are `vs`.

The meanings of the modifier characters are:

`v` Give verbose output. When used with the option characters `d`, `r`, `q`, or `m`, this gives a verbose file-by-file description of the making of a new archive file from the old archive (if one exists) and the constituent files. When used with `x`, this precedes each file with its name.

`c` Suppress the message that is produced by default when the archive file `afile` is created.

`l` Place temporary files in the local current working directory, rather than in the directory specified by the environment variable `TMPDIR` or in the default directory.

`s` Force the regeneration of the archive symbol table even if `ar` is not invoked with a command which will modify the archive file contents. This command is useful to restore the archive symbol table after it has been stripped [see STRIP(SD_CMD)].

SEE ALSO

LD(SD_CMD), STRIP(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

awk — pattern-directed scanning and processing language

SYNOPSIS

```
awk [ -Fc ] [ -f progfile ] [ 'program' ] [
parameters ] [ file ... ]
```

DESCRIPTION

The `awk` command executes programs written in the awk programming language, which is specialized for data manipulation. An awk program is a sequence of patterns and corresponding actions. When input is read that matches a pattern, the action associated with that pattern is carried out.

The `file` arguments contain the input to be read. If no files are given or the filename - is given, the standard input is used.

Each line of input is matched in turn against the set of patterns in the program. The awk program may either be in a file `progfile` or may be specified in the command line as a string enclosed in single quotes.

Each line of input is matched in turn against each pattern in the program. For each pattern matched, the associated action is executed.

The `awk` command interprets each input line as a sequence of fields where, by default, a field is a string of non-blank, non-tab characters. This default whitespace field delimiter can be changed by using the `-Fc` option, or the variable `FS`; see below. The command `awk` denotes the first field in a line $1, the second $2, and so forth. $0 refers to the entire line. Setting any other field causes the re-evaluation of $0.

Pattern-action statements in an awk program have the form:

pattern { action }

In any pattern-action statement, either the pattern or the action may be omitted. A missing action means print the input line to the standard output; a missing pattern is always matched, and its associated action is executed for every input line read.

Patterns

Patterns are *special patterns* or arbitrary Boolean combinations (!, ||, &&, and parentheses) of regular expressions and relational expressions. The operator ! has the highest precedence, then && and then ||. Evaluation is left to right and stops when truth or falsehood has been determined.

Boolean Operator	Meaning
!	negation
&&	and
||	or

Special Patterns

The `awk` command recognizes two special patterns, **BEGIN** and **END**. **BEGIN** is matched once and its associated action executed before the first line of input is read. **END** is matched once and its associated action executed after the last line of input has been read. (See examples 4 and 5.) These two patterns must have associated actions.

Relational Expressions

A pattern may be any expression that compares strings of characters or numbers. A relational expression is either an

expression relational-operator expression

or an

expression matching-operator regular-expression

The six relational operators are shown in the table below; regular expression matching operators are described later. In a comparison, if both operands are numeric, a numeric comparison is made, otherwise, a string comparison is made.

Relational Operator	**Meaning**
<	less than
<=	less than or equal to
>	greater than
>=	greater than or equal to
!=	not equal to
==	equal to

Regular Expressions

A regular expression must be surrounded by slashes. If `re` is a regular expression, then the pattern

```
/re/
```

matches any line of input that contains a substring specified by the regular expression. A regular expression comparison may be limited to a specific field by one of the two regular expression matching operators, and **!**.

```
$4  /re/
```

matches any line in with the 4th field matches the regular expression `/re/`.

```
$4 !  /re/
```

matches any line in which the 4th field *does not* match the regular expression `/re/`

Regular expressions recognized by `awk` are those recognized by the `ed` [see ED(BU_CMD)] except for \(and \) and with the addition of the special characters +, ?, |, and ( ). Below is a summary of regular expressions recognized by `awk`. The special meaning of a special character can be turned off by preceding the character with a \. The special characters *, + and ? have the highest precedence, then concatenation, then alternation. All are left associative.

Regular Expression	Pattern Matched
c	the character *c* where *c* is not a special character.
\c	the character *c* where *c* is any character.
^	the beginning of the string being compared.
$	the end of the string being compared.
.	any character in the input but newline.
[s]	any character in the set *s* where *s* is a sequence of characters and/or a range of characters, *c-c*.
[^s]	any character not in the set *s*, where *s* is defined as above.
r*	zero or more successive occurrences of the regular expression *r*.
r+	one or more successive occurrences of the regular expression *r*.
r?	zero or one occurrence of the regular expression *r*.
(r)	the regular expression *r*. (Grouping)
rx	the occurrence of regular expression *r* followed by the occurrence of regular expression *x*. (Concatenation)
r|x	the occurrence of regular expression *r* or the occurrence of regular expression *x*.

Pattern Ranges

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

Variables and Special Variables

Variables may be used in an `awk` program by assigning to them. They do not need to be declared. Like *field* variables, all variables are treated as string variable unless used in a clearly numeric context (see **Relational Expressions**). Field variables are designated by a $ followed by a number or numerical expression. New field variables may be created by assigning a value to them. Other special variables set by `awk` are shown in the table below.

Special Variable	Meaning
`$n`	The string read as field n.
`FS`	Input field separator. Set to whitespace by default.
`FILENAME`	Name of the current input file.
`NF`	Number of fields in the current record.
`NR`	Ordinal number of the current record from the start of input.
`OFMT`	*Print* statement output format for numbers. %.6g by default.
`OFS`	*Print* statement output field separation. One blank by default.
`ORS`	*Print* statement output record separator. Newline by default.

Actions

An action is a sequence of statements. A statement can be one of the following. Square brackets indicate optional elements. Keywords are shown in bold font.

if (*expression*) *statement* [**else** *statement*]
while (*expression*) *statement*
for (*expression* ; *expression* ; *expression*) *statement*
break
continue
{ [*statement*] ... }
variable = *expression*
print [*expression-list*] [>*expression*]
printf *format* [, *expression-list*] [>*expression*]
next
exit (*expression*)

Any single statement may be replaced by a statement list enclosed in curly braces. The statements in a statement list are separated by newlines or semicolons. The symbol # anywhere in a program line begins a comment, with is terminated by the end of the line.

Statements are terminated by semicolons, newlines, or right braces. A long statement may be split across several lines by ending each partial line with a \. An empty expression-list stands for the whole input line. Expressions take on string or numeric values as appropriate, and are built using the operators + (addition), - (subtraction), * (multiplication), / (division), % (modulus operator), and concatenation (indicated by a blank between strings in an expression). The C language operators ++, --, +=, -=, *=, /=, and %= are also available in expressions. Variables may be scalars, array elements (denoted x[i]) or fields. Variables are initialized to the null string. Array subscripts may be any

string, not necessarily numeric.

String constants are surrounded by double quotes ("..."). A string expression is created by concatenating constants, variables, field names, array elements, functions and other expressions.

The *expression* acting as the conditional in an **if** statement can include the relational operators, the regular expression matching operators, logical operators, juxtaposition for concatenation and parentheses for grouping. *Expression* is evaluated and if it is non-zero and non-null, *statement* is executed, otherwise if **else** is present, the statement following the **else** is executed.

The **while, for, break,** and **continue** statements are as in the C language.

The **print** statement prints its arguments on the standard output (or on a file if *>expr* is present), separated by the current output field separator (see variable `OFS` below), and terminated by the output record separator (see variable `ORS` below). The **printf** statement formats its expression list according to *format* [see PRINTF(BA_LIB)].

The **next** statement causes the next input line to be scanned, skipping the remaining characters on the current input line. The **exit** statement causes the termination of the `awk` program, skipping the rest of the input.

The built-in function **length**(*s*) returns the length of its arguments taken as a string, or of the whole line, $0, if there is no argument. There are also built-in functions **exp**(*x*) (the exponential function of *x*), **log**(*x*) (natural logarithm of *x*), **sqrt**(*x*) (square root of *x*), and **int**(*x*) (truncates its argument to an integer). **substr**(*s, p, n*) returns the at most *n*-character substring of *s* that begins at position p.

The function **sprintf**(*fmt, expr, expr, ...*) formats the expressions according to the PRINTF(BA_LIB) format given by *fmt* and returns the resulting string.

EXAMPLES

The following are examples of simple `awk` programs:

Print on the standard output all input lines for which field 3 is greater than 5.

```
$3 > 5
```

Print every 10th line.

```
(NR % 10) == 0
```

Print any line with a substring matching the regular expression.

```
/(G|D)(2[0-9][a-zA-Z]*)/
```

Print the second to the last and the last field in each line. Separate the fields by a colon.

```
{OFS=":";print $(NF-1), $NF}
```

Print the line number and number of fields in each line. The 3 strings representing the line number, the colon and the number of fields are concatenated and that string is printed.

```
BEGIN {line = 0}
    {line = line + 1
    print line ":" NF}
```

Print lines longer than 72 characters.

```
length > 72
```

Print first two fields in opposite order separated by the OFS.

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

Add up first column, print sum and average.

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

Print fields in reverse order:

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

Print all lines between occurrences of the strings "start" and "stop":

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

Print all lines whose first field is different from the previous one:

```
$1 != prev { print; prev = $1 }
```

Print file, filling in page number starting at 5:

```
/Page/ { $2 = n++; }
    { print }
```

```
command line:  awk -f program n=5 input
```

USAGE

General.

There are no explicit conversions between numbers and strings. To force an expression to be treated as a number add 0 to it; to force it to be treated as a string concatenate the null string ("") to it.

LEVEL

Level 1.

NAME

banner — make large letters

SYNOPSIS

`banner strings`

DESCRIPTION

The command `banner` prints each argument in large letters (across the page) on the standard output, putting each argument on a separate "line". Spaces can be included in an argument by surrounding it with quotes. The maximum number of characters that can be accomodated in a line is implementation dependent; excess characters are simply ignored.

SEE ALSO

ECHO(BU_CMD)

USAGE

General.

LEVEL

Level 1.

NAME

basename, dirname — deliver portions of path names

SYNOPSIS

```
basename string [ suffix ]
dirname string
```

DESCRIPTION

The command `basename` deletes any prefix ending in / and the `suffix` (if present in `string`) from `string`, and prints the result on the standard output. It is normally used inside substitution marks (``) within command procedures.

The command `dirname` delivers all but the last level of the path name in `string`.

EXAMPLES

The following example moves the named file to a file named `xyz` in the current directory:

```
mv abc `basename /p/q/xyz.c '.c'`
```

The following example will set the variable `NAME` to `/usr/src/cmd`:

```
NAME=`dirname /usr/src/cmd/xyz.c`
```

SEE ALSO

SH(BU_CMD)

USAGE

General.

LEVEL

Level 1.

NAME

cal — print calendar

SYNOPSIS

`cal [ [ month ] year ]`

DESCRIPTION

The `cal` command prints a calendar for the specified year. If a month is also specified, a calendar just for that month is printed. If neither is specified, a calendar for the present month is printed. The argument `year` can be between 1 and 9999. (Note that "`cal` 83" refers to 83 A.D., not 1983.) The `month` is a number between 1 and 12.

USAGE

End-user.

LEVEL

Level 1.

NAME

calendar — reminder service

SYNOPSIS

`calendar`

DESCRIPTION

The command `calendar` consults the file `calendar` in the current directory and prints out lines that contain today's or tomorrow's date anywhere in the line. Month-day date formats such as "Aug. 24," "august 24," "8/24," are recognized. On weekends, "tomorrow" extends through Monday.

USAGE

End-user.

LEVEL

Level 1.

NAME

cat – concatenate and print files

SYNOPSIS

```
cat [ -s ] file ...
```

DESCRIPTION

The command `cat` reads each `file` in sequence and writes it on the standard output. Thus:

```
cat file
```

prints the file, and:

```
cat file1 file2 >file3
```

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

If no input file is given, or if the argument – is encountered, `cat` reads from the standard input file. The `-s` option makes `cat` silent about nonexistent files.

USAGE

General.

Command formats such as

```
cat file1 file2 >file1
```

will cause the original data in `file1` to be lost.

LEVEL

Level 1.

NAME

cd — change working directory

SYNOPSIS

`cd [ directory ]`

DESCRIPTION

If `directory` is not specified, the value of the environmental variable HOME is used as the new working directory. If `directory` specifies a complete path starting with /, ., or .., `directory` becomes the new working directory. If neither case applies, `cd` tries to find the designated directory relative to one of the paths specified by the `CDPATH` environmental variable. `CDPATH` has the same syntax as, and similar semantics to, the `PATH` variable [see SH(BU_CMD)]. The command `cd` must have execute (search) permission in `directory`.

SEE ALSO

PWD(BU_CMD), SH(BU_CMD), CHDIR(BA_OS)

USAGE

General.

LEVEL

Level 1.

NAME

chmod — change mode

SYNOPSIS

`chmod mode files`

DESCRIPTION

The permissions of the named `files` are changed according to `mode`, which may be absolute or symbolic.

An absolute `mode` is a four-octal-digit number constructed from the logical "OR" (sum) of the following modes:

4000	set user ID on execution
2000	set group ID on execution
1000	Reserved
0400	read by owner
0200	write by owner
0100	execute (search in directory) by owner
0040	read by group
0020	write by group
0010	execute (search) by group
0004	read by others
0002	write by others
0001	execute (search) by others

A symbolic `mode` has the form:

[*who*] *op permission* [*op permission*]]

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

The argument *op* 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 argument permission is any combination of the letters `r` (read), `w` (write), `x` (execute), and `s` (set owner or group ID); `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`.

Only the owner of a file (or the super-user) may change its mode. In order to set set-group-ID, the group of the file must correspond to the user's current group ID.

EXAMPLES

The first example denies write permission to others, the second makes a file executable:

```
chmod o-w file
```

```
chmod +x file
```

SEE ALSO

LS(BU_CMD), CHMOD(BA_OS).

USAGE

General.

FUTURE DIRECTIONS

The command `chmod` will be used to specify mandatory locking (enable/disable) on a file. This will be done as follows:

An absolute mode of 20#0 specifies "set-group-ID" if # is 1, 3, 5, or 7; specifies "enable mandatory locking" if # is 0, 2, 4, or 6.

A symbolic mode of `l` specifies mandatory locking, `+` to enable, `-` to disable.

It will not be possible to have set-group-ID set and mandatory locking enabled on a file simultaneously.

LEVEL

Level 1.

NAME

cmp — compare two files

SYNOPSIS

```
cmp [ -l ] [ -s ] file1 file2
```

DESCRIPTION

The command `cmp` compares two files. (If `file1` is `-`, the standard input is used.) 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 identical to the first part of the other, then it is reported that end-of-file was reached in the shorter file (before any differences were found).

Options:

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

`-s` Print nothing for differing files; return codes only.

ERRORS

Exit code 0 is returned for identical files, 1 for different files, and 2 for an inaccessible or missing argument.

SEE ALSO

COMM(BU_CMD), DIFF(BU_CMD)

USAGE

General.

LEVEL

Level 1.

NAME

col — filter reverse line-feeds

SYNOPSIS

`col [ -bfpx ]`

DESCRIPTION

The command `col` reads from the standard input and writes onto the standard output. It performs the line overlays implied by reverse line feeds, and by forward and reverse half-line feeds.

If the `-b` option is given, `col` assumes 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.

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. This treatment can be suppressed by the `-f` (fine) option; in this case, the output from `col` may contain forward half-line feeds, but will still never contain either kind of reverse line motion.

Unless the `-x` option is given, `col` will convert white space to tabs on output wherever possible to shorten printing time.

The ASCII control characters SO and SI 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, newline, SI, SO, VT, reverse line feed, forward half-line feed, and reverse half-line feed. The VT character is an alternate form of full reverse line-feed, included for compatibility with some earlier programs of this type. All other non-printing characters are ignored.

The ASCII codes for the control functions and line-motion sequences mentioned above are as given in the table below. ESC stands for the ASCII "escape" character, with the octal code 033; ESC-*x* means a sequence of two characters, ESC followed by the character *x*.

reverse line feed	ESC-7
reverse half-line feed	ESC-8
forward half-line feed	ESC-9
vertical tab (VT)	013
start-of-text (SO)	016
end-of-text (SI)	017

Normally, `col` will remove any escape sequences found in its input that are unknown to it; the `-p` option may be used to force these to be passed through unchanged. The use of this option is discouraged unless the user is

aware of the consequences.

USAGE

General.

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.

LEVEL

Level 1.

NAME

comm — select or reject lines common to two sorted files

SYNOPSIS

`comm [ -[123] ] file1 file2`

DESCRIPTION

The command `comm` reads `file1` and `file2`, which should be ordered in ASCII collating sequence [see SORT(BU_CMD)], and produces a three-column output: lines only in `file1`; lines only in `file2`; and lines in both files. The file name `-` means the standard input.

Flags 1, 2, or 3 suppress printing of the corresponding column. Thus `comm -12` prints only the lines common to the two files; `comm -23` prints only lines in the first file but not in the second; `comm -123` is a no-op.

SEE ALSO

CMP(BU_CMD), DIFF(BU_CMD), SORT(BU_CMD), UNIQ(BU_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

cp, ln, mv — copy, link or move files

SYNOPSIS

```
cp file1 [ file2 ...] target
ln [ -f ] file1 [ file2 ...] target
mv [ -f ] file1 [ file2 ...] target
```

DESCRIPTION

These commands respectively copy, link, or move files; `file1` and `target` may not be the same. If `target` is not a directory, then only one file may be specified before it; if `target` is an existing file, its contents are destroyed, otherwise (`target` is neither an existing file nor a directory) the file `target` is created. If `target` is a directory, then more than one file may be specified before it; the specified files are respectively copied, linked, or moved to that directory.

cp

If `target` is not a directory, `cp` copies `file1` to `target`. If `target` exists, its contents are overwritten, but the mode, owner, and group are not changed. If `target` is a link to a file, all links remain (the file is changed).

If `target` is a directory, then the specified files are copied to that directory. For each file named, a new file, with the same mode, is created in the target directory; the owner and the group are those of the user making the copy.

ln

If `target` is not a directory, `ln` links `file1` to `target`, that is, the name `target` is linked to the file `file1`. If `target` exists, and its mode forbids writing, the mode is printed, and the user asked for a response; if the response begins with a `y`, (and the user is permitted) then the `ln` occurs. No questions are asked and the `ln` is done where permitted when the `-f` option is used or if the standard input is not a terminal.

If `target` is a directory, then the specified files are linked to that directory. That is, files with the same names are created in the directory, linked to the specified files.

mv

If `target` is not a directory, `mv` moves (renames) `file1` as directed. If `target` does not exist, and has the same parent as `file1`, `file1` may be a directory: this allows a directory rename.

If `target` is a directory, then the specified files are moved to that directory.

If `file1` is a file and `target` is a link to another file with links, the other links remain and `target` becomes a new file.

If `target` is a file, and its mode forbids writing, the mode is printed, and the user asked for a response; if the response begins with a `y`, (and the user is permitted) then the `mv` occurs. No questions are asked and the `mv` is done where permitted when the `-f` option is used or if the standard input is not a terminal.

SEE ALSO

CPIO(BU_CMD), RM(BU_CMD), CHMOD(BU_CMD).

USAGE

General.

If `file1` and `target` lie on different file systems, `mv` may achieve the move by copying the file and deleting the original. In this case any linking relationship with other files is lost.

`ln` will not link across file systems.

LEVEL

Level 1.

NAME

cpio — copy file archives in and out

SYNOPSIS

```
cpio -o[acBv]

cpio -i[Bcdmrtuvf] [patterns]

cpio -p[adlmruv] directory
```

DESCRIPTION

The command `cpio -o` (copy out) reads the standard input to obtain a list of path names and copies those files onto the standard output together with path name and status information. Output is padded to a 512-byte boundary.

The command `cpio -i` (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. The arguments `patterns` are simple regular expressions given in the name-generating notation of the shell [see SH(BU_CMD)]. In `patterns`, meta-characters ?, *, and [...] match the / character. Multiple `patterns` may be specified and if no `patterns` are specified, the default for `patterns` is * (i.e., select all files). The extracted files are conditionally created and copied into the current directory tree based upon the options described below. The permissions of the files will be those of the previous `cpio -o`. The owner and group of the files will be that of the current user unless the user is super-user, which causes `cpio` to retain the owner and group of the files of the previous `cpio -o`.

The command `cpio -p` (pass) reads the standard input to obtain a list of path names of files that are conditionally created and copied into the destination `directory` tree based upon the options described below.

Archives of text files created by `cpio` are portable between implementations of System V.

The meanings of the available options are:

- `a` Reset access times of input files after they have been copied. [When option `-l` (see below) is also specified, the linked files do not have their access times reset.]
- `B` Input/output is to be blocked 5120 bytes to the record (does not apply to the `pass` option; meaningful only with data directed to or from character special files).
- `d` Directories are to be created as needed.
- `c` Write header information in ASCII character form for portability.
- `r` Interactively rename files. If the user types a null line, the file is skipped.
- `t` Print a table of contents of the input. No files are created.
- `u` Copy unconditionally (normally, an older file will not replace a newer file with the same name).

- `v` Verbose: causes the names of the affected files to be printed. With the `t` option, provides a detailed listing.
- `l` Whenever possible, link files rather than copying them. Usable only with the `-p` option.
- `m` Retain previous file modification time. This option is ineffective on directories that are being copied.
- `f` Copy in all files except those in `patterns`.

EXAMPLES

The first example below copies the contents of a directory into an archive; the second duplicates a directory hierarchy:

```
ls | cpio -oc >/dev/mt/0m

cd olddir
find . -depth -print | cpio -pdl newdir
```

SEE ALSO

AR(BU_CMD), FIND(BU_CMD), LS(BU_CMD), TAR(AU_CMD).

USAGE

General.

Only the super-user can copy special files.

LEVEL

Level 1.

NAME

cut – cut out selected fields of each line of a file

SYNOPSIS

```
cut -clist [file1 file2 ...]
cut -flist [-dchar] [-s] [file1 file2 ...]
```

DESCRIPTION

The command `cut` cuts out columns from a table or fields from each line of a file. The fields as specified by `list` can be of fixed length, specified by character position (`-c` option), or the length can vary from line to line and be marked with a field delimiter character like `tab` (`-f` option). The command `cut` can be used as a filter; if no files are given, the standard input is used.

The option qualifier `list` (see options `-c` and `-f` below) is a comma-separated list of integers (in increasing order), with optional `-` to indicate ranges; e.g., `1,4,7`; `1-3,8`; `-5,10` (short for `1-5,10`); or `3-` (short for third through last).

The meanings of the options are:

`-clist` The `list` following `-c` (no space) specifies character positions (e.g., `-c1-72` would pass the first 72 characters of each line).

`-flist` The `list` following `-f` is a list of fields assumed to be separated in the file by a delimiter character (see `-d`); e.g., `-f1,7` copies the first and seventh field only. Lines with no field delimiters will be passed through intact (useful for table subheadings), unless `-s` is specified.

`-dchar` The character following `-d` is the field delimiter (used with the `-f` option only). Default is the tab character. Space or other characters with special meaning to the command interpreter must be quoted.

`-s` Suppresses lines with no delimiter characters when used with the `-f` option. Unless specified, lines with no delimiters will be passed through untouched.

Either the `-c` or the `-f` option must be specified.

EXAMPLES

The following maps user IDs to names:

```
cut -d: -f1,5 /etc/passwd
```

SEE ALSO

GREP(BU_CMD), PASTE(BU_CMD), SH(BU_CMD).

USAGE

General.

Use `grep` to make horizontal "cuts" (by context) through a file, or `paste` to put files together column-wise (i.e., horizontally). To reorder

columns in a table, use `cut` and `paste`.

LEVEL

Level 1.

NAME

date — print or set the date

SYNOPSIS

```
date mmddhhmm[yy]

date [ +format ]
```

DESCRIPTION

The first form of `date` sets the current date and time; it is usable only by the super-user. The first `mm` is the month (number); `dd` is the day (number) of the month; `hh` is the hour (number, 24 hour system); the second `mm` is the minute (number); `yy` is the last 2 digits of the year and is optional. For example:

```
date 10080045
```

sets the date to Oct 8, 12:45 AM. The current year is the default if no year is given. The system operates in GMT; `date` takes care of the conversion to and from local standard and daylight time. (The environment variable TZ specifies the local time-zone; therefore its value affects the conversion between the internal GMT clock and the local time.)

In the second form, if no argument is given, the current date and time are printed. (As above, the environment variable TZ specifies the local time-zone, and therefore its value affects the output.) If an argument beginning with `+` is given, the output of `date` is under the control of the user. The format for the output is similar to that of the first argument to the `printf` routine [see PRINTF(BA_LIB)]. 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.

Field Descriptors:

`n`	insert a newline character
`t`	insert a tab character
`m`	month of year — 01 to 12
`d`	day of month — 01 to 31
`y`	last 2 digits of year — 00 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

EXAMPLE

```
date '+DATE: %m/%d/%y%nTIME: %H:%M:%S'
```

generates the output:

DATE: 08/01/76
TIME: 14:45:05

SEE ALSO

PRINTF(BA_LIB).

USAGE

General.

It is a bad practice to change the date while the system is running multi-user.

LEVEL

Level 1.

NAME

df — report free disk space

SYNOPSIS

`df [ -t ] [ file-system ... ]`

DESCRIPTION

The command `df` prints out the free space (in 512-byte units) and the number of free file slots ("inodes") available for on-line file systems. The argument `file-system` may be specified either by device name (e.g., `/dev/dsk/0s1`) or by mounted directory name (e.g., `/usr`). If no `file-system` is specified, the free space on all of the mounted file systems is printed.

The `-t` option causes the total allocated space figures to be reported as well.

USAGE

General.

LEVEL

Level 1.

NAME

diff — differential file comparator

SYNOPSIS

`diff [ -efbh ] file1 file2`

DESCRIPTION

The command `diff` tells what lines must be changed in two files to bring them into agreement. If `file1 (file2)` is `-`, the standard input is used. If `file1 (file2)` is a directory, then a file in that directory with the name `file2 (file1)` is used. The normal output contains lines of these forms:

`n1 a n3,n4`

`n1,n2 d n3`

`n1,n2 c n3,n4`

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

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

The `-b` option causes trailing blanks (spaces and tabs) to be ignored and other strings of blanks to compare equal.

The `-e` option produces a script of **a**, **c**, and **d** commands for the editor `ed`, which will recreate `file2` from `file1`.

The `-f` option produces a similar script, not useful with `ed`, in the opposite order.

Option `-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.

Options `-e` and `-f` are unavailable with the `-h` option.

SEE ALSO

CMP(BU_CMD), COMM(BU_CMD), ED(BU_CMD).

ERRORS

Exit status is:

0 no differences

1 some differences

2 errors

USAGE

General.

Editing scripts produced under the `-e` or `-f` option may be incorrect when dealing with lines consisting of a single period.

LEVEL

Level 1.

NAME

du — estimate file space usage

SYNOPSIS

`du [ -ars ] [ file ... ]`

DESCRIPTION

The command `du` gives an estimate, in 512-byte units, of the file space contained in all the specified files. Whenever a directory is named, all files within it are reported; sub-directories are traversed recursively. If no `file` is specified, the current directory is used.

The option `-s` causes only the grand total (for each of the specified `files`) to be given. The option `-a` causes a report to be generated for each file. With no options, a report is given for each directory only.

`du` is normally silent about directories that cannot be read, files that cannot be opened, etc. The `-r` option will cause `du` to generate messages in such instances.

A file with two or more links is only counted once.

USAGE

General.

If the `-a` option is not used, non-directories given as arguments are not listed.

Files with holes in them may get an incorrect (high) estimate.

LEVEL

Level 1.

NAME

echo — echo arguments

SYNOPSIS

`echo [ arg ] ...`

DESCRIPTION

The command `echo` writes its arguments separated by blanks and terminated by a newline on the standard output. It also understands the following escape conventions.

- `\b` backspace
- `\c` print arguments up to this point, without newline; ignore remainder of command line
- `\f` form-feed
- `\n` newline
- `\r` carriage return
- `\t` tab
- `\v` vertical tab
- `\\` backslash
- `\0`*n* *n* must be a 1-, 2- or 3-digit octal number; specifies the corresponding ASCII character

SEE ALSO

SH(BU_CMD).

USAGE

General.

The command `echo` is useful for producing diagnostics in command scripts and for sending known data into a pipe.

Arguments containing blanks and escape sequences must be enclosed in double quotes.

LEVEL

Level 1.

NAME

ed, red — text editor

SYNOPSIS

```
ed [- ] [ -p string ] [ file ]

red [- ] [ -p string ] [ file ]
```

DESCRIPTION

The command `ed` is a text editor. If the `file` argument is given, `ed` simulates an `e` command (see below) on the named file; that is to say, the file is read into the `ed` buffer so that it can be edited.

The option `-` 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 *!command*.

The `-p` option allows the user to specify a prompt string. (This option is new in System V Release 2.)

`ed` 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.

The command `red` is a restricted version of `ed`. It will only allow editing of files in the current directory, and prohibits executing commands via *!command*. Attempts to bypass these restrictions result in an error message.

Commands to `ed` have a simple and regular structure: zero, one, or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses can very often 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.

`ed` supports a limited form of *regular expression* notation; regular expressions are used in addresses to specify lines and in some commands (e.g., *s*) to specify portions of a line that are to be substituted. A regular expression (RE) 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 RE*s match a **single** character:

1.1 An ordinary character (**not** one of those discussed in 1.2 below) 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 1.4 below).

b. ^ (caret or circumflex), which is special at the *beginning* of an *entire* RE (see 3.1 and 3.2 below), or when it immediately follows the left of a pair of square brackets ([]) (see 1.4 below).

c. $ (currency symbol), which is special at the *end* of an entire RE (see 3.2 below).

d. The character used to bound (i.e., delimit) an entire RE, which is special for that RE (for example, see how slash (/) is used in the g command, below).

1.3 A period (.) is a one-character RE that matches any character except newline.

1.4 A non-empty 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 caret (^), 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); e.g., **[]a-f]** matches either a right square bracket (]) or one of the letters **a** through **f** inclusive. The four characters listed in 1.2.a above stand for themselves within such a string of characters.

The following rules may be used to construct *RE*s from one-character REs:

2.1 A one-character RE is a 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 leftmost string that permits a match is chosen.

2.3 A one-character RE followed by \{*m*\}, \{*m*,\}, or \{*m*,*n*\} is a 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 A RE enclosed between the character sequences \(and \) is a 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 circumflex (^) 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 (e.g., //) is equivalent to the last RE encountered.

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. 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 lower-case letter. Lines are marked with the *k* command described below.
5. A 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 including the current line, so that the entire buffer is searched.
6. A 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 including the current line.

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; e.g, **−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 rule 8 immediately above, the address − refers to the line preceding the current line. 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 a comma (,). They may also be separated by a semicolon (;). 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. above). 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. However, any command (except `e`, `f`, `r`, or `w`) may be suffixed by `l`, `n`, or `p` in which case the current line is either listed, numbered or printed, respectively, as discussed below under the `l`, `n`, and `p` commands.

`( . )a`
<text>
.

The append command reads the given text and appends it after the addressed line; the **current line** becomes the last inserted line, or, if there were none, 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 file name is given, the currently-remembered file name, if any, is used (see the f command). The number of characters read is typed; the name *file* is remembered for possible use as a default file name in subsequent e, r, and w commands. If *file* is replaced by !, the rest of the line is taken to be a command whose output is to be read. Such a command is *not* remembered as the current file name.

E *file*

The E (edit) 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, the file-name command changes the currently-remembered file name to *file*; otherwise, it prints the currently-remembered file name.

(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. The g, G, v, and V commands are **not** permitted in the *command list*.

(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 DEL or BREAK).

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 lower-case letter. The address 'x then addresses this line; . is unchanged.

(. , .)l

The list command prints the addressed lines in an unambiguous way: a few non-printing characters (e.g., **tab, backspace**) are represented by (hopefully) mnemonic overstrikes. All other non-printing characters are printed in octal, and long lines are folded. An l command may be appended to any other command other than e, f, r, or w.

(. , .)m*a*

The move command repositions the addressed line(s) 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 number 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 other 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 other 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 quit command causes ed to exit.

Q
The editor exits 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 file name is given, the currently-remembered file name, if any, is used (see e and f commands). The currently-remembered file name is **not** changed unless *file* is the very first file name 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 command whose output is to be read. Such a command is **not** remembered as the current file name.

(. , .)s/*RE*/*replacement*/ or
(. , .)s/*RE*/*replacement*/g or
(. , .)s/*RE*/*replacement*/n n = 1-512
The substitute command searches each addressed line for an occurrence of the specified RE. In each line in which a match is found, all (non-overlapped) 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. 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.

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 by \. 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 by \. Such substitution cannot be done as part of a `g` or `v` command list.

`(.,.)t`*a*

This command acts just like the `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 undo command nullifies the effect of 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*

The write command writes the addressed lines into the named file. The currently-remembered file name is **not** changed unless *file* is the very first file name mentioned since `ed` was invoked. If no file name is given, the currently-remembered file name, 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 command whose standard input is the addressed lines. Such a command is **not** remembered as the current file name.

`($)=`

The line number of the addressed line is typed; . is unchanged by this command.

!*command*

The remainder of the line after the ! is sent to the command interpreter to be interpreted as a command. Within the text of that command, the unescaped character % is replaced with the remembered file name; if a ! appears as the first character of the command, it is replaced with the text of the previous command. Thus, ! ! will repeat the last command. If any expansion is performed, the expanded line is echoed; . is unchanged.

(.+1)

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 (BREAK) is sent, `ed` prints a ? and returns to its command level.

If the closing delimiter of a RE or of a replacement string (e.g., /) 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?
```

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 the editor buffer via the `e` or `q` commands. It prints ? and allows the user to continue editing. A second `e` or `q` command at this point will take effect. The − command-line option inhibits this feature.

ERRORS

? for command errors.

?*file* for an inaccessible file.

The `h` and `H` (help) commands for detailed explanations).

FILES

ed.hup work is saved here if the terminal is hung up.

SEE ALSO

GREP(BU_CMD), SED(BU_CMD), SH(BU_CMD).

USAGE

General.

A ! command cannot be subject to a *g* or a *v* command.

The sequence \n in a RE does not match a newline character.

If the editor input is coming from a command file (i.e., ed file < ed-cmd-file), the editor will exit at the first failure of a command that is in the command file.

FUTURE DIRECTIONS

The option - will be replaced by -s, in order to conform to the syntax standard. The old form of the option will continue to be accepted for some time.

LEVEL

Level 1.

NAME

expr — evaluate expression

SYNOPSIS

`expr expression`

DESCRIPTION

The command `expr` evaluates an expression and writes the result on the standard output. Terms of the expression must be separated by blanks. Characters special to the command interpreter must be escaped. Note that **0** is returned to indicate a zero value, rather than the null string. Strings containing blanks or other special characters should be quoted. Integer-valued arguments may be preceded by a unary minus sign.

The operators 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. The symbol *arg* represents an argument.

arg \| *arg*
: returns the first *arg* if it is neither null nor **0**, otherwise returns the second *arg*.

arg \& *arg*
: returns the first *arg* if neither *arg* is null or **0**, otherwise returns **0**.

arg { =, \>, \>=, \<, \<=, != } *arg*
: returns the result of an integer comparison if both arguments are integers, otherwise returns the result of a lexical comparison.

arg { +, − } *arg*
: addition or subtraction of integer-valued arguments.

arg { *, /, % } *arg*
: multiplication, division, or remainder of the integer-valued arguments.

arg : *arg*
: The 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`, except that all patterns are "anchored" (i.e., 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

1. `a=`expr $a + 1``

   adds 1 to the variable **a**.

2. For $a equal to either "/usr/abc/file" or just "file"

```
expr  $a  :  '.*/\(.*\)'  \|  $a
```

returns the last segment of a path name (i.e., `file`). Watch out for / alone as an argument: `expr` will take it as the division operator.

3. A better representation of example 2.

```
expr  //$a  :  '.*/\(.*\)'
```

The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression.

4. `expr  $VAR  :  '.*'`

returns the number of characters in $VAR.

SEE ALSO

ED(BU_CMD), SH(BU_CMD).

ERRORS

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.

USAGE

General.

After argument processing [see SH(BU_CMD)], `expr` cannot tell the difference between an operator and an operand except by the value. If **$a** is **=**, 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=
```

LEVEL

Level 1.

NAME

file – determine file type

SYNOPSIS

```
file [ -f lfile ] file ...
```

DESCRIPTION

The command `file` performs a series of tests on each specified `file` in an attempt to classify it. If it appears to be a text file, `file` examines an initial segment and makes a guess about its language. (The answer is not guaranteed to be correct.) If an argument is an executable (**"a.out"**) file it is identified as such, and any other available information is reported.

If the `-f` option is given, the next argument is taken to be a file containing the names of the files to be examined.

USAGE

General.

LEVEL

Level 1.

NAME

find — find files

SYNOPSIS

`find path-name-list expression`

DESCRIPTION

The command `find` recursively descends the directory hierarchy for each path name in the `path-name-list` (i.e., one or more path names) seeking files that match a boolean `expression` written in the primaries given below. In the following descriptions, the argument *n* is used as a decimal integer where *+n* means more than *n*, *−n* means less than *n* and *n* means exactly *n*.

The `expression` argument is made up of:

`−name` *file* — True if *file* matches the current file name. The argument syntax of `sh` [see SH(BU_CMD)] may be used if escaped (especially note **[**, **?** and ***)**).

`−perm`*onum* — True if the file permission flags exactly match the octal number *onum* [see CHMOD(BU_CMD)] If *onum* is prefixed by a minus sign, more flag bits (017777) [see STAT(BA_OS)] become significant and the flags are compared.

`−type` *c* — True if the type of the file is *c*, where *c* is **b**, **c**, **d**, **p**, or **f** for block special file, character special file, directory, fifo (named pipe), or plain file respectively.

`−links`*n* — True if the file has *n* links.

`−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 taken as a user ID.

`−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 taken as a group ID.

`−size` *n*[**c**] — True if the file is *n* "blocks" long (block = 512 bytes). If *n* is followed by a **c**, the size is in characters.

`−atime` *n* — True if the file has been accessed in *n* days. The access time of directories in *path-name-list* is changed by `find` itself.

`−mtime` *n* — True if the file has been modified in *n* days.

`−ctime` *n* — True if the file inode has been changed in *n* days.

`−exec` *cmd* — True if the executed *cmd* returns a zero value as exit status. The end of *cmd* must be punctuated by an escaped semicolon. A command argument `{}` is replaced by the

current path name.

`-ok` *cmd* Like `-exec` except that the generated command line is printed with a question mark first, and is executed only if the user responds by typing **y**.

`-print` Always true; causes the current path name to be printed.

`-newer` *file* True if the current file has been modified more recently than the argument *file*.

`-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 can be useful when `find` is used with `cpio` [see CPIO(BU_CMD)] to transfer files that are contained in directories without write permission.

(*expression*) True if the parenthesized expression is true (parentheses must be escaped if they are special to the command interpreter).

The primaries may be combined using the following operators (in order of decreasing precedence):

1) The negation of a primary (**!** 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).

EXAMPLE

To remove all files named `tmp` or ending in `.xx` that have not been accessed for a week:

```
find / \( -name tmp -o -name '*.xx' \) -atime +7
-exec rm {} \;
```

FILES

```
/etc/passwd
/etc/group
```

SEE ALSO

CHMOD(BU_CMD), CPIO(BU_CMD), SH(BU_CMD), TEST(BU_CMD), STAT(BA_OS).

USAGE

General.

LEVEL

Level 1.

NAME

grep — search a file for a pattern

SYNOPSIS

```
grep [ options ] expression [ files ]
```

DESCRIPTION

The command `grep` searches the input `files` (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The patterns are limited regular `expressions` in the style of `ed`.

The following options are recognized:

- `-v` All lines but those matching are printed.
- `-c` Only a count of matching lines is printed.
- `-i` Ignore upper/lower case distinction during comparisons. (This option is new in System V Release 2.)
- `-l` Only the names of files with matching lines are listed (once), separated by newlines.
- `-n` Each line is preceded by its relative line number in the file.
- `-s` The error messages produced for nonexistent or unreadable files are suppressed.

In all cases, the file name is output if there is more than one input file. Care should be taken when using characters in `expression` that may also be meaningful to the command interpreter. It is safest to enclose the entire `expression` argument in single quotes '...'.

ERRORS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

SEE ALSO

ED(BU_CMD), EGREP(AU_CMD), SED(BU_CMD).

USAGE

General.

FUTURE DIRECTIONS

The functionality of `egrep` and `fgrep` [see EGREP(AU_CMD)] will eventually be provided in `grep`, and those two commands discontinued.

LEVEL

Level 1.

NAME

kill — send signal to a process

SYNOPSIS

`kill [ -signal ] PID ...`

DESCRIPTION

The command `kill` sends the specified `signal` to the specified processes (or process groups). If process number 0 is specified, all processes in the process group are signaled. Process numbers can be found by using `ps` [see PS(BU_CMD)].

The argument `signal` must be specified as a numeric value; these values are implementation dependent. (See **FUTURE DIRECTIONS** below.)

If no signal is specified, `kill` sends SIGTERM (terminate). This will normally kill processes that do not catch or ignore the signal.

The specified process(es) must belong to the user unless the user is superuser.

Further details are described in KILL(BA_OS).

SEE ALSO

PS(BU_CMD), KILL(BA_OS), SIGNAL(BA_OS).

USAGE

General.

FUTURE DIRECTIONS

The command `kill` will be changed to use symbolic names rather than numeric values of signals. The old form will continue to be accepted for some time.

LEVEL

Level 1.

NAME

line — read one line

SYNOPSIS

`line`

DESCRIPTION

The command `line` copies one line (up to a newline) from the standard input and writes it on the standard output. It returns an exit code of 1 on EOF and always prints at least a newline. It is often used within command scripts to read from the user's terminal.

SEE ALSO

SH(BU_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

ls — list contents of directory

SYNOPSIS

`ls [ options ] [ file ... ]`

DESCRIPTION

For each `file`, if it is directory `ls` lists the contents of the directory; if it is a file, `ls` repeats its name and gives any other information requested. The output is sorted alphabetically by default. When no `files` are specified, the current directory is listed. When several arguments are given, the arguments are first sorted appropriately, but files appear before directories and their contents.

Note that the following options are new in System V Release 2: `-C`, `-F`, `-R`, `-m`, `-n`, `-q`, and `-x`.

There are three major listing formats. The default format is to list one entry per line; the options `-C` and `-x` enable multi-column formats; and the `-m` option enables stream output format in which files are listed across the page, separated by commas.

In order to determine output formats for the `-C`, `-x`, and `-m` options, `ls` uses the environmental variable `COLUMNS` to determine the number of character positions available on one output line. If this variable is not set, the `terminfo` database is used to determine the number of columns, based on the environmental variable `TERM`. If this information cannot be obtained, 80 columns is assumed.

There are a large number of options:

`-C` Multi-column output with entries sorted down the columns.

`-F` Put a slash (/) after each filename if that file is a directory and put an asterisk (*) after each filename if that file is executable.

`-R` Recursively list subdirectories encountered.

`-a` List all entries; usually entries whose names begin with a period (.) are not listed.

`-c` Use time of last modification of the i-node (file created, mode changed, etc.) for sorting (`-t`) or printing (`-l`).

`-d` If an argument is a directory, list only its name (not its contents); often used with `-l` to get the status of a directory.

`-f` Force each argument to be interpreted as a directory and list the name found in each slot. This option turns off `-l`, `-t`, `-s`, and `-r`, and turns on `-a`; the order is the order in which entries appear in the directory.

`-g` The same as `-l`, except that the owner is not printed.

`-i` For each file, print the inode number in the first column of the report.

`-l` List in long format, giving mode, number of links, owner, group, size in bytes, and time of last modification for each file (see below). If the file is a special file, the size field will instead contain the major and minor device numbers rather than a size.

`-m` Stream output format; files are listed across the page, separated by commas.

`-n` The same as `-l`, except that the owner's **UID** and group's **GID** numbers are printed, rather than the associated character strings.

`-o` The same as `-l`, except that the group is not printed.

`-p` Put a slash (/) after each filename if that file is a directory.

`-q` Force non-printing characters (in file names) to be displayed as the character (?).

`-r` Reverse the order of sort to get reverse alphabetic or oldest first as appropriate.

`-s` Give size of each file in 512-byte units.

`-t` Sort by time modified (latest first) instead of by name.

`-u` Use time of last access instead of last modification for sorting (with the `-t` option) or printing (with the `-l` option).

`-x` Multi-column output with entries sorted across rather than down the page.

The mode printed under the `-l` option consists of 10 characters that are interpreted as described below.

The first character is:

d if the entry is a directory;
b if the entry is a block special file;
c if the entry is a character special file;
p if the entry is a fifo (named pipe) special file;
- if the entry is an ordinary file.

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

The permissions are indicated as follows:

r if the file is readable;
w if the file is writable;
x if the file is executable (also see below);

– if the indicated permission is *not* granted.

The group-execute permission character is given as **s** if the file has set-group-ID mode; likewise, the user-execute permission character is given as **s** if the file has set-user-ID mode. These are given as **S** (capitalized) if the corresponding execute permission is NOT set. (See, however, **FUTURE DIRECTIONS** below.)

FILES

/etc/passwd	to get user IDs for `ls -l` and `ls -o`.
/etc/group	to get group IDs for `ls -l` and `ls -g`.
/usr/lib/terminfo/*/*	*terminfo* terminal information database

SEE ALSO

CHMOD(BU_CMD), FIND(BU_CMD).

USAGE

General.

FUTURE DIRECTIONS

The group execute permission will be shown as **l** if mandatory locking is enabled for the file. It will not be possible to set-group-ID without also turning on group execute permission; therefore the group execute permission character will have one of the following values: **–**, **x**, **s**, or **l**; (**S** will not be possible).

LEVEL

Level 1.

NAME

mail, rmail – send or read mail

SYNOPSIS

```
mail [ -epqr ] [ -f file ]
mail [ -t ] name ...
rmail [ -t ] name ...
```

DESCRIPTION

The command `mail` without arguments prints a user's mail, message-by-message, in last-in first-out order. For each message, the user is prompted with a **?**, and a line is read from the standard input to determine the disposition of the message:

`<newline>`	Go on to next message.
`+`	Same as <newline>.
`d`	Delete message and go on to next message.
`p`	Print message again.
`–`	Go back to previous message.
`s` [*file*]	Save message in the named *file* (`mbox` is default).
`w` [*file*]	Save message, without its header, in the named *file* (`mbox` is default).
`m` [*name* ...]	Mail the message to the named users (names are user login names; the default is the user).
`q`	Put undeleted mail back in the *mailfile* and stop.
`EOF`	(Usually control-D.) Same as **q**.
`x`	Put all mail back in the *mailfile* unchanged and stop.
`!`*command*	Escape to the command interpreter to execute *command*.
`*`	Print a command summary.

The optional arguments alter the printing of the mail:

`-e`	causes mail not to be printed. An exit value of 0 is returned if the user has mail; otherwise, an exit value of 1 is returned.
`-p`	causes all mail to be printed without prompting for disposition.
`-q`	causes `mail` to terminate after interrupts. Normally an interrupt only causes the termination of the message being printed.
`-r`	causes messages to be printed in first-in, first-out order.
`-f`*file*	causes `mail` to use *file* (e.g., `mbox`) instead of the default *mailfile*.

When `names` (user login names) are given, `mail` takes the standard input up to an end-of-file (or up to a line consisting of just a .) and adds it to each user's *mailfile*. The message is preceded by the sender's name and a postmark. Lines in the message that begin with the word "From" are preceded with a >. The **-t** option causes the message to be preceded by all users the `mail` is sent to. If a user being sent mail is not recognized, or if `mail` is interrupted during input, the file `dead.letter` will be saved to allow editing and resending. Note that this is regarded as a temporary file in that

it is recreated every time needed, erasing the previous contents of `dead.letter`.

To denote a recipient on a remote system, `name` is the user's login name prefixed by the system name and an exclamation mark. Everything after the first exclamation mark is interpreted by the remote system. In particular, if `name` contains additional exclamation marks, it can denote a sequence of machines through which the message is to be sent on the way to its ultimate destination. For example, specifying **a!b!cde** as a recipient's name causes the message to be sent to user **b!cde** on system **a.** System **a** will interpret that destination as a request to send the message to user **cde** on system **b.** This might be useful, for instance, if the sending system can access system **a** but not system **b,** and system **a** has access to system **b.**

The *mailfile* may also contain the first line:

Forward to *person*

which will cause all mail sent to the owner of the *mailfile* to be forwarded to *person.* This is especially useful to forward all of a person's mail to one machine in a multiple machine environment. In order for forwarding to work properly the *mailfile* should have "mail" as group ID, and the group permission should be read-write.

The command `rmail` only permits the sending of mail.

FILES

`/etc/passwd`	to identify sender and locate persons
`$HOME/mbox`	saved mail
`dead.letter`	unmailable text

USAGE

General.

LEVEL

Level 1.

NAME

mkdir — make a directory

SYNOPSIS

`mkdir dirname ...`

DESCRIPTION

The command `mkdir` creates the specified directories. Standard entries, ., for the directory itself, and .., for its parent, are made automatically.

The command `mkdir` requires write permission in the parent directory.

ERRORS

The command `mkdir` returns exit code 0 if all directories were successfully made; otherwise, it prints a diagnostic and returns non-zero.

SEE ALSO

RM(BU_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

nl — line numbering filter

SYNOPSIS

```
nl [-htype] [-btype] [-ftype] [-vstart#] [-iincr]
[-p]  [-lnum]  [-ssep]  [-wwidth]  [-nformat]
[-ddelim] [ file ]
```

DESCRIPTION

The command `nl` reads lines from the named `file` or the standard input if no `file` is named and reproduces the lines on the standard output. Lines are numbered on the left in accordance with the command options in effect.

`nl` views the text it reads in terms of logical pages. Line numbering is reset at the start of each logical page. A logical page consists of a header, a body, and a footer section. Empty sections are valid. Different line numbering options are independently available for header, body, and footer (e.g., no numbering of header and footer lines while numbering blank lines only in the body).

The start of logical page sections are signaled by input lines containing nothing but the following delimiter character(s):

Line	*Start of*
\:\:\:	header
\:\:	body
\:	footer

Unless otherwise specified, `nl` assumes the text being read is in a single logical page body.

Options may appear in any order and may be intermingled with an optional file name. Only one file may be named. The options are:

`-btype` Specifies which logical page body lines are to be numbered. Recognized `types` and their meaning are: **a**, number all lines; **t**, number lines with printable text only; **n**, no line numbering; **`pstring`**, number only lines that contain the regular expression specified in `string`. Default `type` for logical page body is **t** (text lines numbered).

`-htype` Same as `-btype` except for header. Default `type` for logical page header is **n** (no lines numbered).

`-ftype` Same as `-btype` except for footer. Default for logical page footer is **n** (no lines numbered).

`-p` Do not restart numbering at logical page delimiters.

`-vstart#` The initial value used to number logical page lines. Default is **1**.

`-iincr` The increment value used to number logical page lines. Default is **1**.

`-ssep` The character(s) used in separating the line number and the corresponding text line. Default `sep` is a tab.

`-wwidth` The number of characters to be used for the line number. Default `width` is **6**.

`-nformat` The line numbering format. Recognized values are: **ln**, left justified, leading zeroes suppressed; **rn**, right justified, leading zeroes supressed; **rz**, right justified, leading zeroes kept. Default `format` is **rn** (right justified).

`-lnum` The number of blank lines to be considered as one. For example, **-l2** results in only the second adjacent blank being numbered (if the appropriate **-ha**, **-ba**, and/or **-fa** option is set). Default is **1**.

`-dxx` The delimiter characters specifying the start of a logical page section may be changed from the default characters (\:) to two user-specified characters. If only one character is entered, the second character remains the default character (:). No space should appear between the **-d** and the delimiter characters. To enter a backslash, use two backslashes.

EXAMPLE

The command:

```
nl -v10 -i10 -d!+ file1
```

will number file1 starting at line number 10 with an increment of ten. The logical page delimiters are !+.

USAGE

General.

SEE ALSO

PR(BU_CMD).

LEVEL

Level 1.

NAME

nohup — run a command immune to hangups and quits

SYNOPSIS

`nohup command [ arguments ]`

DESCRIPTION

The command `nohup` executes `command` with the signals SIGHUP and SIGQUIT ignored. If output is not re-directed by the user, both standard output and standard error are sent to `nohup.out`. If `nohup.out` is not writable in the current directory, output is redirected to `$HOME/nohup.out`.

EXAMPLE

It is frequently desirable to apply `nohup` to pipelines or lists of commands. This can be done only by placing pipelines and command lists in a single file; this procedure can then be executed as `command`, and the `nohup` applies to everything in the file.

USAGE

General.

SEE ALSO

SH(BU_CMD), SIGNAL(BA_OS).

LEVEL

Level 1.

NAME

pack, pcat, unpack – compress and expand files

SYNOPSIS

```
pack [ - ] [ -f ] name ...

pcat name ...

unpack name ...
```

DESCRIPTION

The command `pack` attempts to store the specified files in a compressed form. Wherever possible (and useful), each input file `name` is replaced by a packed file `name.z` with the same access modes, access and modified dates, and owner as those of `name`. The option **-f** will force packing of `name`. This is useful for causing an entire directory to be packed even if some of the files will not benefit. If `pack` is successful, `name` will be removed. Packed files can be restored to their original form using `unpack` or `pcat`.

The command `pack` uses Huffman (minimum redundancy) codes on a byte-by-byte basis. If the – argument is used, an internal flag is set that causes the number of times each byte is used, its relative frequency, and the code for the byte to be printed on the standard output. Additional occurrences of – in place of `name` will cause the internal flag to be set and reset.

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree forms the first part of each file, it is usually not worthwhile to pack files smaller than three blocks, unless the character frequency distribution is very skewed, which may occur with printer plots or pictures.

Typically, text files are reduced to 60-75% of their original size. Load modules, which use a larger character set and have a more uniform distribution of characters, show little compression, the packed versions being about 90% of the original size.

The command `pack` returns a value that is the number of files that it failed to compress.

No packing will occur if:

the file appears to be already packed;
the file name has more than {NAME_MAX}-2 characters;
the file has links;
the file is a directory;
the file cannot be opened;
the file is empty;
no disk storage blocks will be saved by packing;
a file called `name.z` already exists;
the **.z** file cannot be created;

an I/O error occurred during processing.

The last segment of the file name must contain no more than {NAME_MAX}-2 characters to allow space for the appended **.z** extension.

The command `pcat` does for packed files what `cat` does for ordinary files, except that `pcat` cannot be used as a filter. The specified files are unpacked and written to the standard output. Thus to view a packed file named `name.z` use:

```
pcat name.z    (or pcat name)
```

To make an unpacked copy, called `abc`, of a packed file named `name.z` (without destroying `name.z`) use the command:

```
pcat name >nnn
```

The command `pcat` returns the number of files it was unable to unpack. Failure may occur if:

the file name (exclusive of the **.z**) has more than {NAME_MAX}-2 characters;
the file cannot be opened;
the file does not appear to be the output of `pack`.

The command `unpack` expands files created by `pack`. For each file `name` specified in the command, a search is made for a file called `name.z` (or just `name`, if `name` ends in **.z**). If this file appears to be a packed file, it is replaced by its expanded version. The new file has the **.z** suffix stripped from its name, and has the same access modes, access and modification dates, and owner as those of the packed file.

The command `unpack` returns a value that is the number of files it was unable to unpack. Failure may occur for the same reasons that it may in `pcat`, as well as for the following:

a file with the "unpacked" name already exists;
the unpacked file cannot be created.

USAGE

General.

SEE ALSO

CAT(BU_CMD).

LEVEL

Level 1.

NAME

paste — merge same lines of several files or subsequent lines of one file

SYNOPSIS

```
paste file1 file2 ...
paste -dlist file1 file2 ...
paste -s [-dlist] file1 file2 ...
```

DESCRIPTION

In the first two forms, `paste` concatenates corresponding lines of the given input files `file1`, `file2`, etc. The file-name `-` means standard input. It treats each file as a column or columns of a table and pastes them together horizontally (parallel merging). In the last form above (`-s` option), `paste` combines subsequent lines of the input file (serial merging).

In all cases, lines are glued together with the `tab` character, unless the `-d` option is used (see below).

Output is to the standard output, so that `paste` can be used as the start of a pipe, or as a filter, if `-` is used in place of a file name.

Without the `-d` option, the newline characters of each but the last file (or last line in case of the `-s` option) are replaced by a `tab` character.

When this option is used, a character from the `list` immediately following `-d` replaces the default `tab` as the line concatenation character. The list is used circularly, i.e., when exhausted, it is reused. In parallel merging (i.e., no `-s` option), the lines from the last file are always terminated with a newline character, not from the `list`. The list may contain the special escape sequences: \n (newline), \t (tab), \\ (backslash), and \0 (empty string, not a null character). Quoting may be necessary, if characters have special meaning to the command interpreter.

EXAMPLES

```
ls | paste - - - -
```
list directory in four columns

```
paste -s -d"\t\n" file
```
combine pairs of lines into lines

USAGE

General.

SEE ALSO

CUT(BU_CMD), GREP(BU_CMD), PR(BU_CMD).

LEVEL

Level 1.

NAME

pg – file perusal filter for soft-copy terminals

SYNOPSIS

```
pg [-number] [-p string] [-cefns] [+linenumber]
[+/pattern/] [files...]
```

DESCRIPTION

The command `pg` is a filter that allows the examination of `files` one screenful at a time on a soft-copy terminal. (The file name – and/or null arguments indicate that `pg` should read from the standard input.) Each screenful is followed by a prompt. If the user types a carriage return, another page is displayed; other possibilities are enumerated below.

This command is different from previous paginators in that it allows the user to back up and review something that has already passed. The method for doing this is explained below.

In order to determine terminal attributes, `pg` scans the `terminfo` data base for the terminal type specified by the environmental variable TERM. If TERM is not defined, the terminal type `dumb` is assumed.

The command line options are:

`-number`
: An integer specifying the size (in lines) of the window that `pg` is to use instead of the default. (On a terminal containing 24 lines, the default window size is 23).

`-p string`
: Causes `pg` to use `string` as the prompt. If the prompt string contains a "`%d`", the first occurrence of "`%d`" in the prompt will be replaced by the current page number when the prompt is issued. The default prompt string is ``:''.

`-c` Home the cursor and clear the screen before displaying each page. This option is ignored if **clear_screen** is not defined for this terminal type in the `terminfo` data base.

`-e` Causes `pg not` to pause at the end of each file.

`-f` Normally, `pg` splits lines longer than the screen width, but some sequences of characters in the text being displayed (e.g., escape sequences for underlining) generate undesirable results. The `-f` option inhibits `pg` from splitting lines.

`-n` Normally, commands must be terminated by a newline character. This option causes an automatic end of command as soon as a command letter is entered.

`-s` Causes `pg` to print all messages and prompts in standout mode (usually inverse video).

`+linenumber`
Start up at `linenumber`.

`+/pattern/`
Start up at the first line containing the regular expression pattern.

The responses that may be typed when `pg` pauses can be divided into three categories: those causing further perusal, those that search, and those that modify the perusal environment.

Commands which cause further perusal normally take a preceding address, an optionally signed number indicating the point from which further text should be displayed. This address is interpreted in either pages or lines depending on the command. A signed address specifies a point relative to the current page or line, and an unsigned address specifies an address relative to the beginning of the file. Each command has a default address that is used if none is provided.

The perusal commands and their defaults are as follows:

(+1)`<newline>` or `<blank>`
This causes one page to be displayed. The address is specified in pages.

(+1) **l**
With a relative address this causes `pg` to simulate scrolling the screen, forward or backward, the number of lines specified. With an absolute address this command prints a screenful beginning at the specified line.

(+1) **d** or **^D**
Simulates scrolling half a screen forward or backward.

The following perusal commands take no address.

. or **^L**
Typing a single period causes the current page of text to be redisplayed.

$
Displays the last windowful in the file. Use with caution when the input is a pipe.

The following commands are available for searching for text patterns in the text. The regular expressions described in ED(BU_CMD) are available. They must always be terminated by a `<newline>`, even if the `-n` option is specified.

`i/pattern/`
Search forward for the `ith` (default `i=1`) occurrence of `pattern`. Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

`i^pattern^`
`i?pattern?`
Search backwards for the `ith` (default `i=1`) occurrence of `pattern`. Searching begins immediately before the current page and

continues to the beginning of the current file, without wrap-around. (The ^ notation is useful for terminals that do not properly handle the ?.)

After searching, `pg` will normally display the line found at the top of the screen. This can be modified by appending **m** or **b** to the search command to leave the line found in the middle or at the bottom of the window from now on. The suffix **t** can be used to restore the original situation.

The user of `pg` can modify the environment of perusal with the following commands:

`in` Begin perusing the `ith` next file in the command line. The `i` is an unsigned number, default value is 1.

`ip` Begin perusing the `ith` previous file in the command line. `i` is an unsigned number, default is 1.

`iw` Display another window of text. If `i` is present, set the window size to `i`.

`s filename`
Save the input in the named file. Only the current file being perused is saved. The white space between the **s** and `filename` is optional. This command must always be terminated by a `newline`, even if the `-n` option is specified.

`h` Help by displaying an abbreviated summary of available commands.

`q or Q`
Quit `pg`.

!command
The argument `command` is passed to the command interpreter, whose name is taken from the `SHELL` environmental variable. If this is not available, the default command interpreter is used. This command must always be terminated by a `newline`, even if the `-n` option is specified.

At any time when output is being sent to the terminal, the user can hit the QUIT key (normally control-\) or the interrupt (BREAK) key. This causes `pg` to stop sending output, and to display the prompt. The user may then enter one of the above commands in the normal manner. Unfortunately, some output is lost when this is done, due to the fact that any characters waiting in the terminal's output queue are flushed when the quit signal occurs.

If the standard output is not a terminal, `pg` acts just like the `cat` command, except that a header is printed before each file (if there is more than one).

FILES

`/usr/lib/terminfo/*/*` *terminfo* terminal information database

USAGE

End-user.

While waiting for terminal input, `pg` responds to BREAK, DEL, and QUIT by terminating execution. Between prompts, however, these signals interrupt `pg`'s current task and place the user in prompt mode. These signals should be used with caution when input is being read from a pipe, since an interrupt is likely to terminate the other commands in the pipeline.

If terminal tabs are not set every eight positions, undesirable results may occur.

When `pg` is used as a filter with another command that changes the terminal I/O options, terminal settings may not be restored correctly.

SEE ALSO

ED(BU_CMD), GREP(BU_CMD).

LEVEL

Level 1.

New in System V Release 2.

NAME

pr — print files

SYNOPSIS

`pr [ options ] [ files ]`

DESCRIPTION

The command `pr` prints the named files on the standard output. If `file` is `-`, or if no files are specified, the standard input is assumed. By default, the listing is separated into pages, each headed by the page number, a date and time, and the name of the file.

By default, columns are of equal width, separated by at least one space; lines which do not fit are truncated. If the `-s` option is used, lines are not truncated and columns are separated by the separation character.

If the standard output is associated with a terminal, error messages are withheld until `pr` has completed printing.

The below `options` may appear singly, or may be combined in any order:

`+k` Begin printing with page `k` (default is 1).

`-k` Produce `k-column` output (default is 1). This option should not be used with `-m`. The options `-e` and `-i` are assumed for multi-column output.

`-a` Print multi-column output across the page. This option is appropriate only with the `-k` option.

`-m` Merge and print all files simultaneously, one per column (overrides the `-k option`).

`-d` Double-space the output.

`-eck` Expand `input` tabs to character positions `k+1`, `2*k+1`, `3*k+1`, etc. If `k` is 0 or is omitted, default tab settings at every eighth position are assumed. Tab characters in the input are expanded into the appropriate number of spaces. If `c` (any non-digit character) is given, it is treated as the input tab character (default for `c` is the tab character).

`-ick` In `output`, replace white space wherever possible by inserting tabs to character positions `k+1`, `2*k+1`, `3*k+1`, etc. If `k` is 0 or is omitted, default tab settings at every eighth position are assumed. If `c` (any non-digit character) is given, it is treated as the output tab character (default for `c` is the tab character).

`-nck` Provide `k-digit` line numbering (default for `k` is 5). The number occupies the first `k+1` character positions of each column of normal output or each line of `-m` output. If `c` (any non-digit character) is given, it is appended to the line number to separate it from whatever follows (default for `c` is a tab).

`-wk` Set the width of a line to `k` character positions for multi-column output (default is 72).

`-ok` Offset each line by `k` character positions (default is 0). The number of character positions per line is the sum of the width and offset.

`-lk` Set the length of a page to `k` lines (default is 66). If `k` is less than what is needed for the page header and trailer, then the option `-t` is in effect; that is, header and trailer lines are suppressed in order to make room for text.

`-h header`
Use header as the header to be printed instead of the file name.

`-p` Pause before beginning each page if the output is directed to a terminal (`pr` will ring the bell at the terminal and wait for a carriage return).

`-f` Use form-feed character for new pages (default is to use a sequence of line-feeds). Pause before beginning the first page if the standard output is associated with a terminal.

`-r` Print no diagnostic reports on failure to open files.

`-t` Print neither the five-line identifying header nor the five-line trailer normally supplied for each page. Quit printing after the last line of each file without spacing to the end of the page.

`-sc` Separate columns by the single character `c` instead of by the appropriate number of spaces (default for `c` is a tab).

EXAMPLES

- Print `file1` and `file2` as a double-spaced, three-column listing headed by "file list":

```
pr -3dh "file list" file1 file2
```

- Write `file1` on `file2`, expanding tabs to columns 10, 19, 28, ... :

```
pr -e9 -t <file1 >file2
```

USAGE

General.

LEVEL

Level 1.

NAME

ps — report process status

SYNOPSIS

`ps [ options ]`

DESCRIPTION

The command `ps` prints certain information about active processes. Without `options`, information is printed about processes associated with the current terminal. The output consists of a short listing containing only the process-ID, terminal identifier, cumulative execution time, and the command name. Otherwise, the information that is displayed is controlled by the selection of `options`.

The `options` using lists as arguments can have the list specified in one of two forms: a list of identifiers separated from one another by a comma, or a list of identifiers enclosed in double quotes and separated from one another by a comma and/or one or more spaces.

The `options` are:

`-e` Print information about all processes.

`-d` Print information about all processes, except process group leaders.

`-a` Print information about all processes, except process group leaders and processes not associated with a terminal.

`-f` Generate a `full` listing. (See below for meaning of columns in a full listing).

`-l` Generate a `long` listing. See below.

`-n` *namelist*

The argument will be taken as the name of an alternate system `namelist` file in place of the default.

`-t` *termlist*

Restrict listing to data about the processes associated with the terminals given in `termlist`. Terminal identifiers may be specified in one of two forms: the device's file name (e.g., **tty04**) or if the device's file name starts with **tty**, just the digit identifier (e.g., **04**).

`-p` *proclist*

Restrict listing to data about processes whose process-ID numbers are given in `proclist`.

`-u` *uidlist*

Restrict listing to data about processes whose user-ID numbers or login names are given in `uidlist`. In the listing, the numerical-user-ID will be printed unless the `-f` option is used, in which case the login name will be printed.

`-g` *grplist*

Restrict listing to data about processes whose process group leaders are given in `grplist`.

The column headings and the meaning of the columns in a `ps` listing are given below; the letters **f** and **l** indicate the option (`full` or `long`) that causes the corresponding heading to appear; **all** means that the heading always appears. Note that these two options determine only what information is provided for a process; they do *not* determine which processes will be listed.

F	**(l)**	Flags (octal and additive) associated with the process.
S	**(l)**	The state of the process.
UID	**(f,l)**	The user ID number of the process owner; the login name is printed under the **−f** option.
PID	**(all)**	The process ID of the process; it is possible to kill a process if you know this datum.
PPID	**(f,l)**	The process ID of the parent process.
C	**(f,l)**	Processor utilization for scheduling.
PRI	**(l)**	The priority of the process; higher numbers mean lower priority.
NI	**(l)**	Nice value; used in priority computation.
ADDR	**(l)**	The memory address of the process.
SZ	**(l)**	The size in blocks of the core image of the process.
WCHAN		**(l)** The event for which the process is waiting or sleeping; if blank, the process is running.
STIME		**(f)** Starting time of the process.
TTY	**(all)**	The controlling terminal for the process.
TIME	**(all)**	The cumulative execution time for the process.
CMD	**(all)**	The command name; the full command name and its arguments are printed under the **−f** option.

A process that has exited and has a parent, but has not yet been waited for by the parent, is marked **defunct**.

Under the option `−f`, `ps` tries to determine the command name and arguments given when the process was created by examining memory or the swap area. Failing this, the command name, as it would appear without the option `−f`, is printed in square brackets.

FILES

`/etc/passwd` supplies UID information

USAGE

General.

Things can change while `ps` is running; the snap-shot it gives is only true for an instant, and may not be accurate by the time it is displayed.

LEVEL

Level 1.

NAME

pwd — working directory name

SYNOPSIS

`pwd`

DESCRIPTION

The command `pwd` prints the path name of the working (current) directory.

ERRORS

"Cannot open .." and "Read error in .." indicate possible file system trouble.

USAGE

General.

SEE ALSO

CD(BU_CMD).

LEVEL

Level 1.

NAME

rm, rmdir — remove files or directories

SYNOPSIS

```
rm [ -fri ] file ...

rmdir dir ...
```

DESCRIPTION

The command `rm` removes the entries for one or more files from a directory. If an entry was the last link to the file, the file is destroyed. Removal of a file requires write permission in its directory, but neither read nor write permission on the file itself.

If a file has no write permission and the standard input is a terminal, its permissions are printed and a line is read from the standard input. If that line begins with **y** the file is deleted, otherwise the file remains. No questions are asked when the option `-f` is given or if the standard input is not a terminal.

If a designated file is a directory, an error comment is printed unless the optional argument `-r` has been used. In that case, `rm` recursively deletes the entire contents of the specified directory, and the directory itself.

If the option `-i` (interactive) is in effect, `rm` asks whether to delete each file, and, under `-r`, whether to examine each directory.

The command `rmdir` removes entries for the named directories, which must be empty.

ERRORS

It is forbidden to remove the file .. in order to avoid the consequences of inadvertently doing something like:

```
rm -r .*
```

USAGE

General.

SEE ALSO

UNLINK(BA_OS).

LEVEL

Level 1.

NAME

sed — stream editor

SYNOPSIS

`sed [ -n ] [ -e script ] [ -f sfile ] [ files ]`

DESCRIPTION

The command `sed` copies the named `files` (standard input default) to the standard output, edited according to a script of commands. The `-f` option causes the script to be taken from file `sfile`; these options accumulate. If there is just one `-e` option and no `-f` options, the flag `-e` may be omitted. The `-n` option suppresses the default output. A script consists of editing commands, one per line, of the following form:

[address[, address]] function [arguments]

In normal operation, `sed` cyclically copies a line of input into a *pattern space* (unless there is something left after a `D` command), applies in sequence all commands whose addresses select that pattern space, and at the end of the script copies the pattern space to the standard output (except under `-n`) and deletes the pattern space.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval.

An address is either a decimal number that counts input lines cumulatively across files, a `$` that addresses the last line of input, or a context address, i.e., a `/regular expression/` in the style of the `ed` command modified as follows:

In a context address, the construction `\?regular expression?`, where `?` is any character, is identical to `/regular expression/`. Note that in the context address `\xabc\xdefx`, the second `x` stands for itself, so that the regular expression is `abcxdef`.

The escape sequence `\n` matches a newline embedded in the pattern space.

A period `.` matches any character except the terminal newline of the pattern space.

A command line with no addresses selects every pattern space.

A command line with one address selects each pattern space that matches the address.

A command line with two addresses selects the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line is selected.) Thereafter the process is repeated, looking again for the first address.

Editing commands can be applied only to non-selected pattern spaces by use of the negation function `!` (below).

In the following list of functions the maximum number of permissible addresses for each function is indicated in parentheses.

The argument `text` consists of one or more lines, all but the last of which end with \ to hide the newline. Backslashes in text are treated like backslashes in the replacement string of an s command, and may be used to protect initial blanks and tabs against the stripping that is done on every script line. The argument `rfile` or the argument `wfile` must terminate the command line and must be preceded by exactly one blank. Each `wfile` is created before processing begins. There can be at most 10 distinct `wfile` arguments.

(1) `a\`
`text`
: Append. Place `text` on the output before reading the next input line.

(2) `b label`
: Branch to the : command bearing the `label`. If `label` is empty, branch to the end of the script.

(2) `c\`
`text`
: Change. Delete the pattern space. With 0 or 1 address or at the end of a 2-address range, place `text` on the output. Start the next cycle.

(2) `d`
: Delete the pattern space. Start the next cycle.

(2) `D`
: Delete the initial segment of the pattern space through the first newline. Start the next cycle.

(2) `g`
: Replace the contents of the pattern space by the contents of the hold space.

(2) `G`
: Append the contents of the hold space to the pattern space.

(2) `h`
: Replace the contents of the hold space by the contents of the pattern space.

(2) `H`
: Append the contents of the pattern space to the hold space.

(1) `i\`
`text`
: Insert. Place `text` on the standard output.

(2) `l`
: List the pattern space on the standard output in an unambiguous form. Non-printing characters are spelled in two-digit ASCII and long lines are folded.

(2) `n`
: Copy the pattern space to the standard output. Replace the pattern space with the next line of input.

(2) `N`
: Append the next line of input to the pattern space with an embedded newline. (The current line number changes.)

(2) `p`
: Print. Copy the pattern space to the standard output.

(2) `P`
: Copy the initial segment of the pattern space through the first newline to the standard output.

(1) `q`
: Quit. Branch to the end of the script. Do not start a new cycle.

(2) `r rfile`
: Read the contents of `rfile`. Place them on the output before reading the next input line.

(2) s/regular expression/replacement/flags
Substitute the replacement string for instances of the regular expression in the pattern space. Any character may be used instead of /. For a fuller description see ED(BU_CMD). The value of flags is zero or more of:

- n n= 1 - 512. Substitute for just the n th occurrence of the regular expression.
- g Global. Substitute for all nonoverlapping instances of the regular expression rather than just the first one.
- p Print the pattern space if a replacement was made.
- w wfile
  Write. Append the pattern space to wfile if a replacement was made.

(2) t label Test. Branch to the : command bearing the label if any substitutions have been made since the most recent reading of an input line or execution of a t. If label is empty, branch to the end of the script.

(2) w wfile Write. Append the pattern space to wfile.

(2) x Exchange the contents of the pattern and hold spaces.

(2) y/string1/string2/
Transform. Replace all occurrences of characters in string1 with the corresponding character in string2. The lengths of string1 and string2 must be equal.

(2)! function
Don't. Apply the function (or group, if function is {) only to lines *not* selected by the address(es).

(0): label This command does nothing; it bears a label for b and t commands to branch to.

(1) = Place the current line number on the standard output as a line.

(2) { Execute the following commands through a matching } only when the pattern space is selected.

(0) An empty command is ignored.

(0) # If a # appears as the first character on the first line of a script file, then that entire line is treated as a comment, with one exception. If the character after the # is an 'n', then the default output will be suppressed. The rest of the line after #n is also ignored. A script file must contain at least one non-comment line.

USAGE

General.

SEE ALSO

AWK(BU_CMD), ED(BU_CMD), GREP(BU_CMD).

LEVEL

Level 1.

NAME

sh, rsh — shell, the standard/restricted command interpreter

SYNOPSIS

```
sh [ flags ] [ args ]
rsh [ flags ] [ args ]
```

DESCRIPTION

The command `sh` is a command interpreter that executes commands read from a terminal or a file. The command `rsh` is a restricted version of the standard command interpreter `sh`; it is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. See **Invocation** below for the meaning of flags and other arguments to the shell.

Definitions

A *blank* is a tab or a space.

A *name* is a sequence of letters, digits, or underscores beginning with a letter or underscore.

A *parameter* is a name, a digit, or any of the characters *, @, #, ?, −, $, and !.

Commands

A *simple-command* is a sequence of non-blank *words* separated by *blanks*. The first word specifies the name of the command to be executed. Except as specified below, the remaining words are passed as arguments to the invoked command. The command name is passed as argument 0 [see EXEC(BA_OS)]. The *value* of a *simple-command* is its exit status if it terminates normally, or (octal) 200+status if it terminates abnormally [see SIGNAL(BA_OS) for a list of status values].

A *pipeline* is a sequence of one or more commands separated by |. The standard output of each command but the last is connected by a "pipe" [see PIPE(BA_OS)] to the standard input of the next command. Each command is run as a separate process; the shell waits for the last command to terminate. The exit status of a pipeline is the exit status of the last command.

A *list* is a sequence of one or more pipelines separated by ;, &, &&, or ||, and optionally terminated by ; or &. Of these four symbols, ; and & have equal precedence, which is lower than that of && and ||. The symbols && and || also have equal precedence. The symbol ; causes sequential execution of the preceding pipeline; the symbol & causes asynchronous execution of the preceding pipeline (i.e., the shell does **not** wait for that pipeline to finish). The symbol && (||) causes the *list* following it to be executed only if the preceding pipeline returns a zero (non-zero) exit status. An arbitrary number of newlines may appear in a *list*, instead of semicolons, to delimit commands.

A command is either a *simple-command* or one of the following. Unless otherwise stated, the value returned by a command is that of the

last *simple-command* executed in the command.

`for` *name* [`in` *word* ...] `do` *list* `done`
Each time a `for` command is executed, *name* is set to the next *word* taken from the `in` *word* list. If `in` *word* ... is omitted, then the `for` command executes the `do` *list* once for each positional parameter that is set (see **Parameter Substitution** below). Execution ends when there are no more words in the list.

`case` *word* `in` [*pattern* [| *pattern*] ...) *list* ;;] ... `esac`
A `case` command executes the *list* associated with the first *pattern* that matches *word*. The form of the patterns is the same as that used for file-name generation (see **File Name Generation**) except that a slash, a leading dot, or a dot immediately following a slash need not be matched explicitly.

`if` *list* `then` *list* [`elif` *list* `then` *list*] ... [`else` *list*] `fi`
The *list* following `if` is executed and, if it returns a zero exit status, the *list* following the first `then` is executed. Otherwise, the *list* following `elif` is executed and, if its value is zero, the *list* following the next `then` is executed. Failing that, the `else` *list* is executed. If no `else` *list* or `then` *list* is executed, then the `if` command returns a zero exit status.

`while` *list* `do` *list* `done`
A `while` command repeatedly executes the `while` *list* and, if the exit status of the last command in the list is zero, executes the `do` *list*; otherwise the loop terminates. If no commands in the `do` *list* are executed, then the `while` command returns a zero exit status; `until` may be used in place of `while` to negate the loop termination test.

(*list*)
Execute *list* in a sub-shell.

{ *list*; }
list is simply executed. (The semi-colon may be replaced by a newline.)

`name` ( ) { *list*; }
(New in System V Release 2.) Define a function which is referenced by `name`. The body of the function is the *list* of commands between { and }. (The semi-colon may be replaced by a newline.) Execution of functions is described below (see **Execution**).

The following words are only recognized as the first word of a command and when not quoted:

```
if then else elif fi case esac for
esac for while until do done { }
```

Comments

A word beginning with # causes that word and all the following characters up to a newline to be ignored.

Command Substitution

The standard output from a command enclosed in a pair of grave accents (``) may be used as part or all of a word; trailing newlines are removed.

Parameter Substitution

The character `$` is used to introduce substitutable parameters. There are two types of parameters, positional and keyword. If the parameter name is a single digit (0 - 9), it is a positional parameter; otherwise, the name must be a legal *name* as defined above, and gives a keyword parameter. Positional parameters may be assigned values by `set`. Keyword parameters (also known as variables) may be assigned values by writing:

`name=`*value* [`name=`*value*] ...

Pattern-matching is not performed on *value*. There cannot be a function and a variable with the same `name`.

`$ {`*parameter*`}`
: The value, if any, of the parameter is substituted. The braces are required only when *parameter* is followed by a letter, digit, or underscore that is not to be interpreted as part of its name. If *parameter* is `*` or `@`, all the positional parameters, starting with `$1`, are substituted (separated by spaces). Parameter `$0` is set from argument zero when the shell is invoked.

`$ {`*parameter*`:-`*word*`}`
: If *parameter* is set and is non-null, substitute its value; otherwise substitute *word*.

`$ {`*parameter*`:=`*word*`}`
: If *parameter* is not set or is null set it to *word*; the value of the parameter is substituted. Positional parameters may not be assigned to in this way.

`$ {`*parameter*`:?`*word*`}`
: If *parameter* is set and is non-null, substitute its value; otherwise, print *word* and exit from the shell. If *word* is omitted, the message "parameter null or not set" is printed.

`$ {`*parameter*`:+`*word*`}`
: If *parameter* is set and is non-null, substitute *word*; otherwise substitute nothing.

In the above, *word* is not evaluated unless it is to be used as the substituted string, so that, in the following example, `pwd` is executed only if `d` is not set or is null:

```
echo ${d:-`pwd`}
```

If the colon (`:`) is omitted from the above expressions, the shell only checks whether *parameter* is set or not.

The following parameters are automatically set by the shell:

- `#` The number of positional parameters in decimal.
- `-` Flags supplied to the shell on invocation or by the `set` command.
- `?` The decimal value returned by the last synchronously executed command.
- `$` The process number of this shell.
- `!` The process number of the last background command invoked.

The following parameters are used by the shell:

`HOME`
: The default argument (home directory) for the `cd` command.

`PATH`
: The search path for commands (see **Execution** below). The user may not change `PATH` if executing under `rsh`.

`CDPATH`
: The search path for the `cd` command. The syntax and usage is similar to that of `PATH`.

`MAIL`
: If this parameter is set to the name of a mail file, then the shell informs the user of the arrival of mail in the specified file. In System V Release 2, the user is informed only if `MAIL` is set and `MAILPATH` is not set.

`MAILCHECK`
: (New in System V Release 2.) This parameter specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the `MAILPATH` or `MAIL` parameters. The default value is 600 seconds (10 minutes). If set to 0, the shell will check before each primary prompt.

`MAILPATH`
: (New in System V Release 2.) The symbol : separated list of file names. If this parameter is set, the shell informs the user of the arrival of mail in any of the specified files. Each file name can be followed by % and a message that will be printed when the modification time changes. The default message is "`you have mail`".

`PS1`
: Primary prompt string, by default "`$ `".

`PS2`
: Secondary prompt string, by default "`> `".

`IFS`
: Internal field separators, normally *space*, *tab*, and *newline*.

`SHACCT`
: (New in System V Release 2.) If this parameter is set to the name of a file writable by the user, the shell will write

an accounting record in the file for each shell procedure executed.

`SHELL`
(New in System V Release 2.) When the shell is invoked, it scans the environment (see **Environment** below) for this name. If it is found and there is an 'r' in the file name part of its value, the shell becomes a restricted shell.

The shell gives default values to `PATH`, `PS1`, `PS2`, `MAILCHECK` and `IFS`.

Blank Interpretation

After parameter and command substitution, the results of substitution are scanned for internal field separator characters (those found in `IFS`) and split into distinct arguments where such characters are found. Explicit null arguments (`""` or `''`) are retained. Implicit null arguments (those resulting from parameters that have no values) are removed.

File Name Generation

Following substitution, each command *word* is scanned for the characters `*`, `?`, and `[`. If one of these characters appears the word is regarded as a *pattern*. The word is replaced with alphabetically sorted file names that match the pattern. If no file name is found that matches the pattern, the word is left unchanged. The character `.` at the start of a file name or immediately following a `/`, as well as the character `/` itself, must be matched explicitly.

`*` Matches any string, including the null string.
`?` Matches any single character.
`[...]` Matches any one of the enclosed characters. A pair of characters separated by `-` matches any character lexically between the pair, inclusive. If the first character following the opening "`[`" is a `!` any character not enclosed is matched.

Quoting

The following characters have a special meaning to the shell and cause termination of a word unless quoted:

`;` `&` `(` `)` `|` `^` `<` `>` *newline space tab*

A character may be **quoted** (i.e., made to stand for itself) by preceding it with a `\`. The pair `\`*newline* is ignored. All characters enclosed between a pair of single quote marks (`''`), except a single quote, are quoted. Inside double quote marks (`""`), parameter and command substitution occurs and `\` quotes the characters `\`, `` ` ``, `"`, and `$`. `"$*"` is equivalent to `"$1 $2 ..."`, whereas `"$@"` is equivalent to `"$1" "$2" ...`

Prompting

When used interactively, the shell prompts with the value of `PS1` before reading a command. If at any time a newline is typed and further input is needed to complete a command, the secondary prompt (i.e., the value of `PS2`) is issued.

Input/Output

Before a command is executed, its input and output may be redirected using a special notation interpreted by the shell. The following may appear anywhere in a *simple-command* or may precede or follow a command and are **not** passed on to the invoked command; substitution occurs before *word* or *digit* is used:

`<`*word*	Use file *word* as standard input (file descriptor 0).
`>`*word*	Use file *word* as standard output (file descriptor 1). If the file does not exist it is created; otherwise, it is truncated to zero length.
`>>`*word*	Use file *word* as standard output. If the file exists output is appended to it (by first seeking to the end-of-file); otherwise, the file is created.
`<<[ - ]`*word*	The shell input is read up to a line that is the same as *word*. or to an end-of-file. The resulting document becomes the standard input. If any character of *word* is quoted, no interpretation is placed upon the characters of the document; otherwise, parameter and command substitution occurs, (unescaped) `\`*newline* is ignored, and `\` must be used to quote the characters `\`, `$`, `` ` ``, and the first character of `word`. If `-` is appended to `<<`, all leading tabs are stripped from *word* and from the document.
`<&`*digit*	Use the file associated with file descriptor *digit* as standard input. Similarly for the standard output using `>&`*digit*.
`<&-`	The standard input is closed. Similarly for the standard output using `>&-`.

If any of the above is preceded by a digit, the file descriptor which will be associated with the file is that specified by the digit (instead of the default 0 or 1). For example:

```
... 2>&1
```

associates file descriptor 2 with the file currently associated with file descriptor 1.

The order in which redirections are specified is significant. The shell evaluates redirections left-to-right. For example:

```
... 1>xxx 2>&1
```

first associates file descriptor 1 with the file *xxx*. It associates file descriptor 2 with the file associated with file descriptor 1 (i.e., *xxx*). If the order of redirections were reversed, file descriptor 2 would be associated with the terminal (assuming file descriptor 1 had been) and file descriptor 1 would be associated with file *xxx*.

If a command is followed by & the default standard input for the command is the empty file `/dev/null`. Otherwise, the environment for the execution of a command contains the file descriptors of the invoking shell as modified by input/output specifications.

Redirection of output is not allowed in the restricted shell.

Environment

The *environment* is a list of name-value pairs that is passed to an executed program in the same way as a normal argument list. The shell interacts with the environment in several ways. On invocation, the shell scans the environment and creates a parameter for each name found, giving it the corresponding value. If the user modifies the value of any of these parameters or creates new parameters, none of these affects the environment unless the `export` command is used to bind the shell's parameter to the environment (see also `set -a`). A parameter may be removed from the environment with the `unset` command (new in System V Release 2). The environment seen by any executed command is thus composed of any unmodified name-value pairs originally inherited by the shell, minus any pairs removed by `unset`, plus any modifications or additions, all of which must be noted in `export` commands.

The environment for any *simple-command* may be augmented by prefixing it with one or more assignments to parameters. Thus:

```
TERM=123 cmd
(export TERM; TERM=123; cmd)
```

(where `cmd` uses the value of the environmental variable `TERM`) are equivalent as far as the execution of `cmd` is concerned.

If the `-k` flag is set, **all** keyword arguments are placed in the environment, even if they occur after the command name. The following first prints `a=b c` and `c`:

```
echo a=b c
set -k
echo a=b c
```

Signals

The INTERRUPT and QUIT signals for an invoked command are ignored if the command is followed by &; otherwise signals have the values inherited by the shell from its parent (but see also the `trap` command below).

Execution

Each time a command is executed, the above substitutions are carried out. If the command name matches one of the **Special Commands** listed below, it is executed in the shell process. If the command name does not match a **Special Command**, but matches the name of a defined function (functions are new in System V Release 2), the function is executed in the shell process (note how this differs from the execution of shell procedures). The positional parameters `$1`, `$2`, ... are set to the arguments of the function. If the command name matches neither a **Special Command** nor the name of a defined function, a new process is created and an attempt is made to execute the command via an `exec` routine [see EXEC(BA_OS)].

The variable `PATH` defines the search path for the directory containing the command. Alternative directory names are separated by a colon (`:`). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If the command name contains a `/` the search path is not used; such commands will not be executed by the restricted shell. Otherwise, each directory in the path is searched for an executable file. If the file has execute permission but is not an executable (**"a.out"**) file, it is assumed to be a file containing shell commands. A sub-shell is spawned to read it. A parenthesized command is also executed in a sub-shell.

The following is new in System V Release 2:

The location in the search path where a command was found is remembered by the shell (to help avoid unnecessary `exec` calls later). If the command was found in a relative directory, its location must be redetermined whenever the current directory changes. The shell forgets all remembered locations whenever the `PATH` variable is changed or the `hash -r` command is executed (see below).

Special Commands

Except as specified, input/output redirection is not permitted for these commands in System V Release 1. In System V Release 2, such redirection is permitted; file descriptor 1 is the default output location.

`:`
: No effect; the command does nothing. A zero exit code is returned.

`. `*file*
: Read and execute commands from *file* and return. The search path specified by `PATH` is used to find the directory containing *file*.

`break [ `*n*` ]`
: Exit from the enclosing `for` or `while` loop, if any. If *n* is specified break *n* levels.

`continue [ `*n*` ]`
: Resume the next iteration of the enclosing `for` or `while` loop.

If *n* is specified resume at the *n*-th enclosing loop.

`cd [ arg ]`

Change the current directory to *arg*. The variable HOME is the default *arg*. The variable CDPATH defines the search path for the directory containing *arg*. Alternative directory names are separated by a colon (`:`). The default path is **null** (specifying the current directory). Note that the current directory is specified by a null path name, which can appear immediately after the equal sign or between the colon delimiters anywhere else in the path list. If *arg* begins with a `/` the search path is not used. Otherwise, each directory in the path is searched for *arg*. The `cd` command may not be executed by `rsh`.

`echo [ arg ... ]`

(Not a Special Command in System V Release 1.) Echo arguments. See ECHO(BU_CMD) for usage and description.

`eval [ arg ... ]`

The arguments are read as input to the shell and the resulting command(s) executed.

`exec [ arg ... ]`

The command specified by the arguments is executed in place of this shell without creating a new process. Input/output arguments may appear and, if no other arguments are given, cause the shell input/output to be modified.

`exit [ n ]`

Causes a shell to exit with the exit status specified by *n*. If *n* is omitted the exit status is that of the last command executed (an end-of-file will also cause the shell to exit.)

`export [ name ... ]`

The given *name*s are marked for automatic export to the *environment* of subsequently-executed commands. If no arguments are given, a list of all names that are exported in this shell is printed. Function names may **not** be exported.

`hash [ -r ] [ name ... ]`

(New in System V Release 2.) For each *name*, the location in the search path of the command specified by *name* is determined and remembered by the shell. The `-r` option causes the shell to forget all remembered locations. If no arguments are given, information about remembered commands is presented.

`pwd`

(Not a Special Command in System V Release 1.) Print the current working directory. See PWD(BU_CMD) for usage and description.

`read [ name ... ]`

One line is read from the standard input and the first word is assigned to the first *name*, the second word to the second *name*, etc., with leftover words assigned to the last *name*. Only the characters in the variable IFS are recognized as delimiters. The

return code is 0 unless an end-of-file is encountered.

`readonly [` *name* ... `]`

The given *name*s are marked **readonly** and the values of the these *name*s may not be changed by subsequent assignment. If no arguments are given, a list of all **readonly** names is printed.

`return [` *n* `]`

(New in System V Release 2.) Causes a function to exit with the return value specified by *n*. If *n* is omitted, the return status is that of the last command executed.

`set [ --aefhkntuvx [` *arg* ... `] ]`

- `-a` (New in System V Release 2.) Mark variables which are modified or created for export.
- `-e` Exit immediately if a command exits with a non-zero exit status.
- `-f` (New in System V Release 2.) Disable file name generation
- `-h` (New in System V Release 2.) Locate and remember function commands as functions are defined (function commands are normally located when the function is executed).
- `-k` All keyword arguments are placed in the environment for a command, not just those that precede the command name.
- `-n` Read commands but do not execute them.
- `-t` Exit after reading and executing one command.
- `-u` Treat unset variables as an error when substituting.
- `-v` Print shell input lines as they are read.
- `-x` Print commands and their arguments as they are executed.
- `--` Do not change any of the flags; useful in setting `$1` to `-`.

Using `+` rather than `-` causes these flags to be turned off. These flags can also be used upon invocation of the shell. The current set of flags may be found in `$-`. The remaining arguments are positional parameters and are assigned, in order, to `$1`, `$2`, ... If no arguments are given the values of all names are printed.

`shift [` *n* `]`

The positional parameters from $n+1$ ... are renamed `$1` .... If *n* is not given, it is assumed to be 1.

`test`

Evaluate conditional expressions. See TEST(BU_CMD) for usage and description.

`times`

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

`trap [ arg ] [ n ] ...`

The command *arg* is to be read and executed when the shell receives signal(s) *n*. (Note that *arg* is scanned once when the trap is set and once when the trap is taken.) Trap commands are executed in order of signal number. Any attempt to set a trap on a signal that was ignored on entry to the current shell is ineffective. If *arg* is absent all trap(s) *n* are reset to their original values. If *arg* is the null string this signal is ignored by the shell and by the commands it invokes. If *n* is 0 the command *arg* is executed on exit from the shell. The `trap` command with no arguments prints a list of commands associated with each signal number.

`type [ name ... ]`

(New in System V Release 2.) For each *name*, indicate how it would be interpreted if used as a command name.

`ulimit [ -f n ]`

If the `-f n` option is used, then a size limit of `n` blocks is imposed on files written by the shell and its child processes (files of any size may be read). If *n* is omitted, the current limit is printed. If no option is given, "`-f`" is assumed.

`umask [ nnn ]`

The user file-creation mask is set to *nnn* [see UMASK(BA_OS)]. If *nnn* is omitted, the current value of the mask is printed.

`unset [ name ... ]`

(New in System V Release 2.) For each *name*, remove the corresponding variable or function. The variables `PATH`, `PS1`, `PS2`, `MAILCHECK` and `IFS` cannot be unset.

`wait [ n ]`

Wait for the specified process and report its termination status. If *n* is not given all currently active child processes are waited for and the return code is zero.

Invocation

If the shell is invoked through an `exec` routine [see EXEC(BA_OS)] and the first character of argument zero is `-`, commands are initially read from `/etc/profile` and from `$HOME/.profile`, if such files exist. Thereafter, commands are read as described below. The flags below are interpreted by the shell on invocation only; note that unless the `-c` or `-s` flag is specified, the first argument is assumed to be the name of a file containing commands, and the remaining arguments are passed as positional parameters to that command file:

`-c` *string* If the `-c` flag is present commands are read from *string*.

`-s` If the `-s` flag is present or if no arguments remain commands are read from the standard input. Any remaining arguments specify the positional parameters. Shell output (except for **Special Commands**) is written to file descriptor 2.

`-i` If the `-i` flag is present or if the shell input and output are attached to a terminal, this shell is *interactive*. In this case TERMINATE is ignored (so that `kill 0` does not kill an interactive shell) and INTERRUPT is caught and ignored (so that `wait` is interruptible). In all cases, QUIT is ignored by the shell.

`-r` If the `-r` flag is present the shell is a restricted shell.

The remaining flags and arguments are described under the `set` command above.

Rsh Only

`rsh` is used to set up login names and execution environments whose capabilities are more controlled than those of the standard shell. The actions of `rsh` are identical to those of `sh`, except that the following are disallowed:

changing directory [see CD(BU_CMD)],
setting the value of `PATH`,
specifying path or command names containing `/`,
redirecting output (`>` and `>>`).

The restrictions above are enforced after `.profile` is interpreted.

When a command to be executed is found to be a shell procedure, `rsh` invokes `sh` to execute it. Thus, it is possible to provide to the end-user shell procedures that have access to the full power of the standard shell, while imposing a limited menu of commands; this scheme assumes that the end-user does not have write and execute permissions in the same directory.

The net effect of these rules is that the writer of the `.profile` has complete control over user actions, by performing guaranteed setup actions and leaving the user in an appropriate directory (probably **not** the login directory).

EXIT STATUS

Errors detected by the shell, such as syntax errors, cause the shell to return a non-zero exit status. If the shell is being used non-interactively execution of the shell file is abandoned. Otherwise, the shell returns the exit status of the last command executed (see also the `exit` command above).

FILES

`/etc/profile`
`$HOME/.profile`
`/dev/null`

USAGE

General.

(Not for System V Release 1.) If a command is executed, and a command with the same name is installed in a directory in the search path before the

directory where the original command was found, the shell will continue to `exec` the original command. Use the `hash` command to correct this situation.

(Not for System V Release 1.) If the current directory or the one above it is moved, `pwd` may not give the correct response. Use the `cd` command with a full path name to correct this situation.

SEE ALSO

CD(BU_CMD), ECHO(BU_CMD), PWD(BU_CMD), TEST(BU_CMD), UMASK(BU_CMD), DUP(BA_OS), EXEC(BA_OS), FORK(BA_OS), PIPE(BA_OS), SIGNAL(BA_OS), SYSTEM(BA_OS), ULIMIT(BA_OS), UMASK(BA_OS), WAIT(BA_OS).

LEVEL

Level 1.

NAME

sleep — suspend execution for an interval

SYNOPSIS

```
sleep time
```

DESCRIPTION

The command `sleep` suspends execution for `time` seconds. It is used to execute a command after a certain amount of time, as in:

```
(sleep 105; command)&
```

or to execute a command every so often, as in:

```
while true
do
        command
        sleep 37
done
```

USAGE

General.

SEE ALSO

ALARM(BA_OS), SLEEP(BA_OS).

LEVEL

Level 1.

NAME

sort — sort and/or merge files

SYNOPSIS

```
sort [-cmu] [-ooutput] [-ykmem] [-zrecsz]
[-dfinr] [-btx] [+pos1 [-pos2]] [files]
```

DESCRIPTION

The command `sort` sorts lines of all the named files together and writes the result on the standard output. The standard input is read if — is used as a file name or no input files are named.

Comparisons are based on one or more sort keys extracted from each line of input. By default, there is one sort key, the entire input line, and ordering is lexicographic by bytes in machine collating sequence.

The following options alter the default behavior:

`-c` Check that the input file is sorted according to the ordering rules; give no output unless the file is out of sort.

`-m` Merge only, the input files are already sorted.

`-u` Unique: suppress all but one in each set of lines having equal keys.

`-ooutput` The argument given is the name of an output file to use instead of the standard output. This file may be the same as one of the inputs. There may be optional blanks between `-o` and `output`.

`-ykmem` The amount of main memory used by the sort has a large impact on its performance. Sorting a small file in a large amount of memory is a waste. If this option is omitted, `sort` begins using a system default memory size, and continues to use more space as needed. If this option is presented with a value, `kmem`, `sort` will start using that number of kilobytes of memory, unless the administrative minimum or maximum is violated, in which case the corresponding extremum will be used. Thus, **-y0** is guaranteed to start with minimum memory. By convention, `-y` (with no argument) starts with maximum memory.

`-zrecsz` The size of the longest line read is recorded in the sort phase so buffers can be allocated during the merge phase. If the sort phase is omitted via the `-c` or `-m` options, a popular system default size will be used. Lines longer than the buffer size will cause `sort` to terminate abnormally. Supplying the actual number of bytes in the longest line to be merged (or some larger value) will prevent abnormal termination.

The following options override the default ordering rules.

`-d` "Dictionary" order: only letters, digits and blanks (spaces and tabs) are significant in comparisons.

`-f` Fold lower case letters into upper case.

`-i` Ignore characters outside the ASCII range 040-0176 in non-numeric comparisons.

`-n` An initial numeric string, consisting of optional blanks, optional minus sign, and zero or more digits with optional decimal point, is sorted by arithmetic value. The `-n` option implies the `-b` option (see below). Note that the `-b` option is only effective when restricted sort key specifications are in effect.

`-r` Reverse the sense of comparisons.

When ordering options appear before restricted sort key specifications, the requested ordering rules are applied globally to all sort keys. When attached to a specific sort key (described below), the specified ordering options override all global ordering options for that key.

The notation `+pos1 -pos2` restricts a sort key to one beginning at `pos1` and ending at `pos2`. The characters at positions `pos1` and `pos2` are included in the sort key (provided that `pos2` does not precede `pos1`). A missing `-pos2` means the end of the line.

Specifying `pos1` and `pos2` involves the notion of a field, a minimal sequence of characters followed by a field separator or a newline. By default, the first blank (space or tab) of a sequence of blanks acts as the field separator. All blanks in a sequence of blanks are considered to be part of the next field; for example, all blanks at the beginning of a line are considered to be part of the first field. The treatment of field separators can be altered using the options:

`-tx` Use `x` as the field separator character; `x` is not considered to be part of a field (although it may be included in a sort key). Each occurrence of `x` is significant (e.g., `xx` delimits an empty field).

`-b` Ignore leading blanks when determining the starting and ending positions of a restricted sort key. If the `-b` option is specified before the first `+pos1` argument, it will be applied to all `+pos1` arguments. Otherwise, the **b** flag may be attached independently to each `+pos1` or `-pos2` argument (see below).

The arguments `pos1` and `pos2` each have the form `m.n` optionally followed by one or more of the flags **bdfinr.** A starting position specified by `+m.n` is interpreted to mean the `n+1st` character in the `m+1st` field. A missing `.n` means **.0,** indicating the first character of the `m+1st` field. If the **b** flag is in effect `n` is counted from the first non-blank in the `m+1st` field; `+m.0b` refers to the first non-blank character in the `m+1st` field.

A last position specified by `-m.n` is interpreted to mean the `nth` character (including separators) after the last character of the `mth` field. A missing

.n means .0, indicating the last character of the mth field. If the **b** flag is in effect n is counted from the last leading blank in the m+1st field; -m.1b refers to the first non-blank in the m+1st field.

When there are multiple sort keys, later keys are compared only after all earlier keys compare equal. Lines that otherwise compare equal are ordered with all bytes significant.

EXAMPLES

Sort the contents of infile with the second field as the sort key:

```
sort +1 -2 infile
```

Sort, in reverse order, the contents of infile1 and infile2, placing the output in outfile and using the first character of the second field as the sort key:

```
sort -r -o outfile +1.0 -1.2 infile1 infile2
```

Sort, in reverse order, the contents of infile1 and infile2 using the first non-blank character of the second field as the sort key:

```
sort -r +1.0b -1.1b infile1 infile2
```

Print the password file sorted by the numeric user ID (the third colon-separated field):

```
sort -t: +2n -3 /etc/passwd
```

Print the lines of the already sorted file infile, suppressing all but the first occurrence of lines having the same third field (the options **-um** with just one input file make the choice of a unique representative from a set of equal lines predictable):

```
sort -um +2 -3 infile
```

ERRORS

Sort comments and exits with non-zero status for various trouble conditions (e.g., when input lines are too long), and for disorder discovered under the -c option.

When the last line of an input file is missing a newline character, sort appends one, prints a warning message, and continues.

USAGE

General.

SEE ALSO

COMM(BU_CMD), JOIN(AU_CMD), UNIQ(BU_CMD).

LEVEL

Level 1.

NAME

spell — find spelling errors

SYNOPSIS

```
spell [ -v ] [ -b ] [ -x ] [ +local_file ] [
files ]
```

DESCRIPTION

The command `spell` collects words from the named `files` and looks them up in a spelling list. Words that neither occur among nor are derivable (by applying certain inflections, prefixes, and/or suffixes) from words in the spelling list are printed on the standard output. If no `files` are named, words are collected from the standard input.

Under the `-v` option, all words not literally in the spelling list are printed, and plausible derivations from the words in the spelling list are indicated.

Under the `-b` option, British spelling is checked. Besides preferring *centre, colour, programme, speciality, travelled,* etc., this option insists upon *-ise* in words like *standardise.*

Under the `-x` option, every plausible stem is printed with `=` for each word.

Under the `+local_file` option, words found in `local_file` are removed from `spell's` output. The argument `local_file` is the name of a user-provided file that contains a sorted list of words, one per line. With this option, the user can specify a set of words that are correct spellings (in addition to `spell's` own spelling list) for each job.

USAGE

End-user.

FUTURE DIRECTIONS

In order to the command syntax standard, the `+local_file` option will be changed to the form `-flocal_file`. The old form will continue to be accepted for some time.

LEVEL

Level 1.

NAME

split — split a file into pieces

SYNOPSIS

`split [ -n ] [ file [ name ] ]`

DESCRIPTION

The command `split` reads `file` and writes it in `n-line` pieces (default 1000 lines) onto a set of output files. The name of the first output file is `name` with **aa** appended, and so on lexicographically, up to **zz** (a maximum of 676 files). The argument `name` cannot be longer than {NAME_MAX}-2 characters. If no output name is given, **x** is default.

If no input file is given, or if - is given in its stead, then the standard input file is used.

USAGE

General.

SEE ALSO

CSPLIT(AU_CMD).

LEVEL

Level 1.

NAME

sum — print checksum and block count of a file

SYNOPSIS

```
sum [ -r ] file
```

DESCRIPTION

The command `sum` calculates and prints a checksum for the named file, and also prints the space used by the file, in 512-byte units. The option `-r` causes an alternate algorithm to be used in computing the checksum.

The algorithms used are uniform across all System V implementations, so that the same checksum is obtained for the same file, independent of the hardware and implementation.

USAGE

General.

LEVEL

Level 1.

NAME

tail — deliver the last part of a file

SYNOPSIS

```
tail [ ±[number][lbc[f] ] ] [ file ]
```

DESCRIPTION

The command `tail` copies the named file to the standard output beginning at a designated place. If no file is named, the standard input is used.

Copying begins at distance `+number` from the beginning, or `-number` from the end of the input (if `number` is null, the value 10 is assumed). The arguments `number` is counted in units of lines, blocks, or characters, according to the appended option `l`, `b`, or `c`. When no units are specified, counting is by lines.

With the `-f` ("follow") option, if the input file is not a pipe, the program will not terminate after the line of the input file has been copied, but will enter an endless loop: it sleeps for a second and then attempts to read and copy further records from the input file. Thus it may be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail -f fred
```

will print the last ten lines of the file `fred`, followed by any lines that are appended to `fred` between the time `tail` is initiated and killed. As another example, the command:

```
tail -15cf fred
```

will print the last 15 characters of the file `fred`, followed by any lines that are appended to `fred` between the time `tail` is initiated and killed.

USAGE

General.

Tails relative to the end of the file are saved in a buffer, and thus are limited in length.

Various kinds of anomalous behavior may happen with character special files.

LEVEL

Level 1.

NAME

tee — join pipes and make copies of input

SYNOPSIS

```
tee [ -i ] [ -a ] [ file ] ...
```

DESCRIPTION

The command `tee` transcribes the standard input to the standard output and makes copies in the `files`. The `-i` option ignores interrupts; the `-a` option causes the output to be appended to the `files` rather than overwriting them.

USAGE

General.

LEVEL

Level 1.

NAME

test — condition evaluation command

SYNOPSIS

```
test expr
[ expr ]
```

DESCRIPTION

The command `test` evaluates the expression `expr` and, if its value is true, returns a zero (true) exit status; otherwise, a non-zero (false) exit status is returned; `test` also returns a non-zero exit status if there are no arguments. The following primitives are used to construct `expr`:

-r `file`	true if `file` exists and is readable.
-w `file`	true if `file` exists and is writable.
-x `file`	true if `file` exists and is executable.
-f `file`	true if `file` exists and is a regular file.
-d `file`	true if `file` exists and is a directory.
-c `file`	true if `file` exists and is a character special file.
-b `file`	true if `file` exists and is a block special file.
-p `file`	true if `file` exists and is a named pipe (fifo).
-u `file`	true if `file` exists and its set-user-ID bit is set.
-g `file`	true if `file` exists and its set-group-ID bit is set.
-s `file`	true if `file` exists and has a size greater than zero.
-t[`fildes`]	true if the open file whose file descriptor number is `fildes` (1 by default) is associated with a terminal device.
-z `s1`	true if the length of string `s1` is zero.
-n `s1`	true if the length of the string `s1` is non-zero.
`s1 = s2`	true if strings `s1` and `s2` are identical.
`s1 != s2`	true if strings `s1` and `s2` are `not` identical.
`s1`	true if `s1` is `not` the null string.
`n1` **-eq** `n2`	true if the integers `n1` and `n2` are algebraically equal. Any of the comparisons **-ne, -gt, -ge, -lt,** and **-le** may be used in place of **-eq.**

These primaries may be combined with the following operators:

!	unary negation operator.
-a	binary `and` operator.

-o binary `or` operator (**-a** has higher precedence than **-o**).

(expr) parentheses for grouping.

Notice that all the operators and flags are separate arguments to `test`. Notice also that parentheses are meaningful to `sh` and, therefore, must be escaped.

In the second form of the command (i.e., the one that uses **[ ]**, rather than the word `test`), the square brackets must be delimited by blanks.

USAGE

General.

SEE ALSO

FIND(BU_CMD), SH(BU_CMD).

LEVEL

Level 1.

NAME

touch — update access and modification times of a file

SYNOPSIS

`touch [ -amc ] [ mmddhhmm[yy] ] file ...`

DESCRIPTION

The command `touch` causes the access and modification times of each `file` to be updated. The `file` is created if it does not exist. If no time is specified the current time is used. The `-a` and `-m` options cause touch to update only the access or modification times respectively (default is `-am`). The `-c` option silently prevents `touch` from creating the `file` if it did not previously exist.

The return code from `touch` is the number of files for which the times could not be successfully modified (including files that did not exist and were not created).

USAGE

General.

LEVEL

Level 1.

NAME

tr — translate characters

SYNOPSIS

```
tr [ -cds ] [ string1 [ string2 ] ]
```

DESCRIPTION

The command `tr` copies the standard input to the standard output with substitution or deletion of selected characters. Input characters found in `string1` are mapped into the corresponding characters of `string2`. Any combination of the options `-cds` may be used:

`-c` Complements the set of characters in `string1` with respect to the universe of characters whose ASCII codes are 001 through 377 octal.

`-d` Deletes all input characters in `string1`.

`-s` Squeezes all strings of repeated output characters that are in `string2` to single characters.

The following abbreviation conventions may be used to introduce ranges of characters or repeated characters into the strings:

`[a-z]` Stands for the string of characters whose ASCII codes run from character `a` to character `z`, inclusive.

`[a*n]` Stands for `n` repetitions of `a`. If the first digit of `n` is `0`, `n` is considered octal; otherwise, `n` is taken to be decimal. A zero or missing `n` is taken to be huge; this facility is useful for padding `string2`.

The escape character \ may be used to remove special meaning from any character in a string. In addition, \ followed by 1, 2, or 3 octal digits stands for the character whose ASCII code is given by those digits.

The following example creates a list of all the words in `file1` one per line in `file2`, where a word is taken to be a maximal string of alphabetics. The strings are quoted to protect the special characters from interpretation by the command interpreter; 012 is the ASCII code for newline.

```
tr -cs "[A-Z][a-z]" "[\012*]" <file1 >file2
```

USAGE

General.

The command `tr` does not handle ASCII NUL in `string1` or `string2`; it always deletes NUL from input.

LEVEL

Level 1.

NAME

true, false — provide truth values

SYNOPSIS

```
true
```

```
false
```

DESCRIPTION

The command `true` does nothing, and returns exit code zero. The command `false` does nothing, and returns a non-zero exit code. They are typically used to construct command procedures. For example,

```
while true
do
    command
done
```

USAGE

General.

SEE ALSO

SH(BU_CMD).

LEVEL

Level 1.

NAME

umask — set file-creation mode mask

SYNOPSIS

`umask [ ooo ]`

DESCRIPTION

The user file-creation mode mask is set to `ooo`. The three octal digits refer to read/write/execute permissions for *owner*, *group*, and *others*, respectively [see CHMOD(BU_CMD)]. The value of each specified digit is subtracted from the corresponding "digit" specified by the system for the creation of a file. For example, `umask 022` removes *group* and *others* write permission (files normally created with mode `777` become mode `755`; files created with mode `666` become mode `644`).

If `ooo` is omitted, the current value of the mask is printed.

USAGE

General.

SEE ALSO

CHMOD(BU_CMD).

LEVEL

Level 1.

NAME

uname — print name of current system

SYNOPSIS

`uname [ -snrvma ]`

DESCRIPTION

The command `uname` prints the current system name on the standard output file. The options cause selected information returned by UNAME(BA_OS) to be printed:

`-s` print the system name (default). This is a name by which the system is known in the local installation.

`-n` print the nodename. The nodename may be a name by which the system is known to a communications network.

`-r` print the operating system release.

`-v` print the operating system version.

`-m` print the machine hardware name.

`-a` print all the above information.

USAGE

General.

SEE ALSO

UNAME(BA_OS).

LEVEL

Level 1.

NAME

uniq — report repeated lines in a file

SYNOPSIS

```
uniq [ -udc [ +n ] [ -n ] ] [ input [ output ] ]
```

DESCRIPTION

The command `uniq` reads the input file comparing adjacent lines. In the normal case, the second and succeeding copies of repeated lines are removed; the remainder is written on the output file. The arguments `input` and `output` should always be different. Note that repeated lines must be adjacent in order to be found [see SORT(BU_CMD)]. If the `-u` flag is used, just the lines that are not repeated in the original file are output. The `-d` option specifies that one copy of just the repeated lines is to be written. The normal mode output is the union of the `-u` and `-d` mode outputs.

The `-c` option supersedes `-u` and `-d` and generates an output report in default style but with each line preceded by a count of the number of times it occurred.

The `n` arguments specify skipping an initial portion of each line in the comparison:

`-n` The first `n` fields together with any blanks before each are ignored. A field is defined as a string of non-space, non-tab characters separated by tabs and spaces from its neighbors.

`+n` The first `n` characters are ignored. Fields are skipped before characters.

USAGE

General.

SEE ALSO

COMM(BU_CMD), SORT(BU_CMD).

LEVEL

Level 1.

NAME

wait — await completion of process

SYNOPSIS

`wait [ pid ]`

DESCRIPTION

With no argument, `wait` waits until all processes started with & have completed, and reports on abnormal terminations. If a numeric argument `pid` is given, and is the process id of a background process, then `wait` waits until that process has completed. Otherwise, if `pid` is not a background process, `wait` waits until all background processes have completed.

USAGE

General.

SEE ALSO

SH(BU_CMD), WAIT(BA_OS).

LEVEL

Level 1.

NAME

wc — word count

SYNOPSIS

```
wc [ -lwc ] [ files ]
```

DESCRIPTION

The command `wc` counts lines, words, and characters in the named files, or in the standard input if no `files` appear. It also keeps a total count for all named files. A word is defined as a maximal string of characters delimited by spaces, tabs, or newlines.

The options `l`, `w`, and `c` may be used in any combination to specify that a subset of lines, words, and characters are to be reported. The default is `-lwc`.

When `files` are specified on the command line, their names will be printed along with the counts.

USAGE

General.

LEVEL

Level 1.

Part III

Advanced Utilities Extension Definition

Chapter 5
Introduction

5.1 OVERVIEW

The Advanced Utilities Extension is intended to be the next expansion step after the Basic Utilities Extension.

The System V Base System and Basic Utilities Extension are prerequisites for this Extension.

5.2 DESCRIPTION

UTILITIES

at +	egrep **	news	uulog
batch +	ex +	od	uuname
cancel	fgrep **	passwd	uupick
chgrp	id	shl +	uustat
chown	join	stty	uuto
cron	logname	su	uux
crontab +	lp	tabs	vi +
csplit	lpstat	tar	wall
cu	mailx +	tty	who
dd	mesg	uucp	write
dircmp	newgrp		

\+ New in System V Release 2.
** Level 2: December 1, 1985.

Chapter 6
Commands and Utilities

NAME

at, batch — execute commands at a later time

SYNOPSIS

```
at time [ date ] [ + increment ]
at -r job ...
at -l [ job ] ...

batch
```

DESCRIPTION

The commands `at` and `batch` read commands from standard input to be executed at a later time. The command `at` allows you to specify when the commands should be executed, while jobs queued with `batch` will execute when system load level permits. The option `-r` removes jobs previously scheduled with `at`. The `-l` option reports all jobs scheduled for the invoking user.

Standard output and standard error output are mailed to the user unless they are redirected elsewhere. The environment variables, current directory, umask, and ulimit are retained when the commands are executed. Open file descriptors, traps, and priority are lost.

Users are permitted to use `at` if their name appears in the file `/usr/lib/cron/at.allow`. If that file does not exist, the file `/usr/lib/cron/at.deny` is checked to determine if the user should be denied access to `at`. If neither file exists, only root is allowed to submit a job. If only `at.deny` exists and is empty, global usage is permitted. The allow/deny files consist of one user name per line.

The `time` may be specified as 1, 2, or 4 digits. One and two digit numbers are taken to be hours, four digits to be hours and minutes. The time may alternately be specified as two numbers separated by a colon, meaning `hour:minute`. A suffix `am` or `pm` may be appended; otherwise a 24-hour clock time is understood. The suffix `zulu` may be used to indicate GMT. The special names `noon`, `midnight`, `now`, and `next` are also recognized.

An optional `date` may be specified as either a month name followed by a day number (and possibly year number preceded by a comma) or a day of the week (fully spelled or abbreviated to three characters). Two special "days", `today` and `tomorrow` are recognized. If no `date` is given, `today` is assumed if the given hour is greater than the current hour and `tomorrow` is assumed if it is less. If the given month is less than the current month (and no year is given), next year is assumed.

The optional `increment` is simply a number suffixed by one of the following: `minutes`, `hours`, `days`, `weeks`, `months`, or `years`. (The singular form is also accepted.)

Thus legitimate commands include:

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

The commands `at` and `batch` write the job number and schedule time to standard error.

The command `batch` submits a batch job. It is almost equivalent to "at now", but not quite. For one, it goes into a different queue. For another, "at now" does not work: it is too late (and results in an error message).

The option `-r` removes jobs previously scheduled by `at` or `batch`. The job number is the number reported at invocation by `at` or `batch`. Job numbers can also be obtained by using the `-l` option. Only the super-user is allowed to remove another user's jobs.

EXAMPLES

The `at` and `batch` commands read from standard input the commands to be executed at a later time. It may be useful to redirect standard output within the specified commands.

This sequence can be used at a terminal:

```
batch
spell filename >outfile
EOT
```

This sequence, which demonstrates redirecting standard error to a pipe, is useful in a command procedure (the sequence of output redirection specifications is significant):

```
batch <<!
spell filename 2>&1 >outfile | mail loginid
!
```

To have a job reschedule itself, `at` can be invoked from within the procedure.

FILES

`/usr/lib/cron/at.allow` - list of allowed users

`/usr/lib/cron/at.deny` - list of denied users

USAGE

General.

SEE ALSO

CRON(AU_CMD).

LEVEL

Level 1.

New in System V Release 2.

NAME

chown, chgrp — change owner or group

SYNOPSIS

```
chown owner file ...

chgrp group file ...
```

DESCRIPTION

The command `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.

The command `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 super-user, the set-user-ID and set-group-ID bits of the file mode will be cleared.

FILES

```
/etc/passwd
/etc/group
```

USAGE

General.

SEE ALSO

CHMOD(BU_CMD), CHOWN(BA_OS).

LEVEL

Level 1.

NAME

cron — clock daemon

SYNOPSIS

`/etc/cron`

DESCRIPTION

The command `cron` executes commands at specified dates and times. Regularly scheduled commands can be specified according to instructions found in crontab files; users can submit their own crontab file via the `crontab` command. Commands which are to be executed only once may be submitted via the `at` command.

A history of all actions taken by cron are recorded in a system log file.

USAGE

Administrator.

Since `cron` never exits, it should only be executed once. This is best done by running it from the initialization process.

SEE ALSO

AT(AU_CMD), CRONTAB(AU_CMD), SH(BU_CMD).

LEVEL

Level 1.

NAME

crontab — user crontab file

SYNOPSIS

```
crontab [file]
crontab -r
crontab -l
```

DESCRIPTION

The command `crontab` copies the specified file, or standard input if no file is specified, into a directory that holds all users' crontabs. The `-r` option removes a user's crontab from the crontab directory. The option `-l` will list the crontab file f the invoking user.

Users are permitted to use `crontab` if their names appear 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 the user should be denied access to `crontab`. If neither file exists, only root is allowed to submit a job. If only `cron.deny` exists and is empty, global usage is permitted. The allow/deny files consist of one user name 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 (0–6 with 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, each one is effective independent of the other. 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).

The sixth field of a line in a crontab file is a string that is executed by the command interpreter 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 command interpreter. The other lines are made available to the command as standard input. The command `cron` supplies a default environment, defining the environmental variables `HOME`, `LOGNAME`, and `PATH`.

NOTE: If standard output and standard error are not redirected, any generated output or errors will be mailed to the user.

FILES

`/usr/lib/cron/cron.allow` list of allowed users

`/usr/lib/cron/cron.deny` list of denied users

USAGE

General.

The new crontab file for a user overwrites an existing one.

SEE ALSO

SH(BU_CMD), CRON(AU_CMD).

LEVEL

Level 1.

New in System V Release 2.

NAME

csplit — context split

SYNOPSIS

```
csplit [-s] [-k] [-fprefix] file arg1 [... argn]
```

DESCRIPTION

The command `csplit` reads `file` and separates it into *n*+1 sections, defined by the arguments `arg1`... `argn`. By default the sections are placed in xx00 ... xxnn (nn 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`.

If the `file` argument is a `-` then standard input is used.

The options to `csplit` are:

`-s` `csplit` normally prints the character counts for each file created. If the `-s` option is present, `csplit` suppresses the printing of all character counts.

`-k` `csplit` normally removes created files if an error occurs. If the `-k` option is present, `csplit` leaves previously created files intact.

`-fprefix`
If the `-f` option is used, the created files are named `prefix00` ... `prefixn`. The default is `xx00` ... `xxn`.

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`. (Regular expressions as in ED(BU_CMD) are accepted.) 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.

`line_no` A file is to be created from the current line up to (but not including) the line number `line_no`. The current line becomes `line_no`.

{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. The command `csplit` does not affect the original file; it is the user's responsibility to remove it.

EXAMPLES

This example creates four files, `cobol00 ... cobol08`:

```
csplit -fcobol file '/procedure division/'
/par5./ /par16./
```

After editing the split files, they can be recombined as follows:

```
cat cobol0[0-3] > file
```

Note that this example overwrites the original file.

This example would split the file at every 100 lines, up to 10,000 lines:

```
csplit -k file  100  {99}
```

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`.

ERRORS

An error is reported if an argument does not reference a line between the current position and the end of the file.

USAGE

General.

SEE ALSO

ED(BU_CMD), SH(BU_CMD).

LEVEL

Level 1.

NAME

cu — call another system

SYNOPSIS

```
cu [-s speed] [-l line] [-h] [-t] [-d] [-o | -e] [-n] telno

cu [-s speed] [-h] [-d] [-o | -e] -l line

cu [-h] [-d] [-o | -e] systemname
```

DESCRIPTION

The command `cu` calls up another system, which will usually be a System V system, but may be a terminal, or a non-System V system. It manages an interactive conversation, with possible transfers of ASCII files.

The third form above, using `systemname`, is new in System V Release 2.

The command `cu` accepts the following options and arguments:

`-sspeed` Specifies the transmission speed. The default value is "Any" speed which will depend on the order of the lines in the system devices file.

`-lline` Specifies a device name to use as the communication line. This can be used to override the search that would otherwise take place for the first available line having the right speed. When the `-l` option is used without the `-s` option, the speed of a line is taken from the devices file. When the `-l` and `-s` options are both used together, `cu` will search the devices file to check if the requested speed for the requested line is available. If so, the connection will be made at the requested speed; otherwise, an error message will be printed and the call will not be made. If the specified device is associated with an auto dialer, a telephone number must be provided. Use of this option with `systemname` rather than `telno` is not allowed (see `systemname` below).

`-h` Emulates local echo, supporting calls to other computer systems which expect terminals to be set to half-duplex mode.

`-t` Used to dial an ASCII terminal which has been set to auto answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

`-d` Causes diagnostic traces to be printed.

`-o` Designates that odd parity is to be generated for data sent to the remote system.

`-e` Designates that even parity is to be generated for data sent to the remote system.

`-n` (New in System V Release 2.) For added security, will prompt the user to provide the telephone number to be dialed rather than taking it from the command line.

`telno` When using an automatic dialer, the argument is the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of 4 seconds.

`systemname` A uucp system name may be used rather than a telephone number; in this case, `cu` will obtain an appropriate direct line or telephone number from a system file.

Note: the `systemname` option should not be used in conjunction with the `-l` and `-s` options as `cu` will connect to the first available line for the system name specified ignoring the requested line and speed.

After making the connection, `cu` runs as two processes: the *transmit* process reads data from the standard input and, except for lines beginning with ~, passes it to the remote system; the *receive* process accepts data from the remote system and, except for lines beginning with ~,passes it to the standard output. Normally, an automatic DC3/DC1 protocol is used to control input from the remote so the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following user initiated commands:

`~.` terminate the conversation.

`~!` escape to an interactive command interpreter on the local system.

`~!cmd...` execute `cmd` on the local system

`~$cmd...` run `cmd` locally and send its output to the remote system for execution.

`~%cd` change the directory on the local system. (New in System V Release 2.)

`~%take from [to]` copy file `from` (on the remote system) to file `to` on the local system. If `to` is omitted, the `from` argument is used in both places.

`~%put from [to]` copy file `from` (on local system) to file `to` on remote system. If `to` is omitted, the `from` argument is used in both places.

`~~line` send the line `~line` to the remote system.

`~%break` transmit a BREAK to the remote system (which can also be specified as `~%b`).

`~%nostop` toggles between DC3/DC1 input control protocol and no input control. This is useful in case the remote

system is one which does not respond properly to the DC3 and DC1 characters.

The *receive* process normally copies data from the remote system to its standard output.

The use of `~%put` requires STTY(AU_CMD) and CAT(BU_CMD) on the remote side. It also requires that the current erase and kill characters on the remote system be identical to these current control characters on the local system. Backslashes are inserted at appropriate places.

The use of `~%take` requires the existence of `echo` and `cat` on the remote system. Also, `tabs` mode [see STTY(AU_CMD)] should be set on the remote system if tabs are to be copied without expansion to spaces.

When `cu` is used on system `X` to connect to system `Y` and subsequently used on system `Y` to connect to system `Z`, commands on system `Y` can be executed by using `~~`. For example, uname can be executed on `Z`, `X`, and `Y` as follows (the response is given in brackets):

```
uname
[ Z ]
~[X]!uname
[ X ]
~~[Y]!uname
[ Y ]
```

In general, `~` causes the command to be executed on the original machine; `~~` causes the command to be executed on the next machine in the chain.

EXAMPLES

To dial a system whose telephone number is 9 1 201 555 1212 using 1200 baud (where dial tone is expected after the 9):

```
cu  -s 1200  9=12015551212
```

If the speed is not specified, "Any" is the default value.

To log in to a system connected by a direct line:

```
cu  -l  /dev/ttyXX
```

or

```
cu -l ttyXX
```

To dial a system with the specific line and a specific speed:

```
cu  -s 1200  -l  ttyXX
```

To dial a system using a specific line associated with an auto dialer:

```
cu  -l  culXX  9=12015551212
```

To use a system name:

```
cu  systemname
```

ERRORS

Exit code is 0 for normal exit, otherwise, −1.

USAGE

End-user.

SEE ALSO

CAT(BU_CMD), ECHO(BU_CMD), STTY(AU_CMD), UNAME(BU_CMD), UUCP(AU_CMD).

LEVEL

Level 1.

NAME

dd — convert and copy a file

SYNOPSIS

```
dd [option=value] ...
```

DESCRIPTION

The command `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.

Option	*Values*
`if=file`	input file name; standard input is default
`of=file`	output file name; standard output is default
`ibs=n`	input block size `n` bytes (default 512)
`obs=n`	output block size (default 512)
`bs=n`	set both input and output block size, superseding `ibs` and `obs`; also, if no conversion is specified, it is particularly efficient since no in-core copy need be done
`cbs=n`	conversion buffer size
`skip=n`	skip `n`" input blocks before starting copy
`seek=n`	seek `n` blocks from beginning of output file before copying
`count=n`	copy only `n`" input blocks
`conv=ascii`	convert **EBCDIC** to **ASCII**
`ebcdic`	convert **ASCII** to **EBCDIC**
`ibm`	slightly different map of **ASCII** to **EBCDIC**
`lcase`	map alphabetics to lower case
`ucase`	map alphabetics to upper case
`swab`	swap every pair of bytes
`noerror`	do not stop processing on an error
`sync`	pad every input block to `ibs`
`...,...`	several comma-separated conversions

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 option `cbs` is used only if `ascii` or `ebcdic` conversion is specified. In the former case `cbs` characters are placed into the conversion buffer, converted to **ASCII**, and trailing blanks trimmed and newline added before sending the line to the output. In the latter case **ASCII** characters are read into the conversion buffer, converted to **EBCDIC**, and blanks added to make up an output block of size `cbs`.

After completion, `dd` reports the number of whole and partial input and output blocks.

EXAMPLE

This command will read an **EBCDIC** tape blocked ten 80-byte **EBCDIC** card images per block into the **ASCII** file `x`:

```
dd if=/dev/rmt/0m of=x ibs=800 cbs=80
conv=ascii,lcase
```

Note the use of raw magtape. The command `dd` is especially suited to I/O on the raw physical devices because it allows reading and writing in arbitrary block sizes.

USAGE

General.

New-lines are inserted only on conversion to **ASCII**; padding is done only on conversion to **EBCDIC**.

LEVEL

Level 1.

NAME

dircmp — directory comparison

SYNOPSIS

```
dircmp [ -d ] [ -s ] dir1 dir2
```

DESCRIPTION

The command `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 the options. If no option is specified, a list is output indicating whether the file names common to both directories have the same contents.

`-d` Compare the contents of files with the same name in both directories and output a list telling what must be changed in the two files to bring them into agreement. The list format is described in DIFF(BU_CMD).

`-s` Suppress messages about identical files.

USAGE

General.

SEE ALSO

CMP(BU_CMD), DIFF(BU_CMD).

LEVEL

Level 1.

NAME

egrep, fgrep — search a file for a pattern

SYNOPSIS

```
egrep [ options ] [ expression ] [ files ]

fgrep [ options ] [ strings ] [ files ]
```

DESCRIPTION

The `egrep` and `fgrep` commands search the input `files` (standard input default) for lines matching a pattern. Normally, each line found is copied to the standard output. The patterns used by `egrep` are full regular `expressions`; `fgrep` patterns are fixed `strings`. The following `options` are recognized:

`-v` All lines but those matching are printed.

`-x` (Exact) only lines matched in their entirety are printed (`fgrep` only).

`-c` Only a count of matching lines is printed.

`-i` Ignore upper/lower case distinction during comparisons.

`-l` Only the names of files with matching lines are listed (once), separated by newlines.

`-n` Each line is preceded by its relative line number in the file.

`-e expression`
(`egrep` only.) Same as a simple `expression` argument, but useful when the `expression` begins with a `-`

`-f file`
The regular `expression` (`egrep`) or `strings` list (`fgrep`) is taken from the `file`.

In all cases, the file name is output if there is more than one input file. Care should be taken when using characters in `expression` that may also be meaningful to the command interpreter. It is safest to enclose the entire `expression` argument in single quotes '...'.

`fgrep` searches for lines that contain one of the `strings` separated by newlines.

`egrep` accepts regular expressions as in ED(BU_CMD), except for `\(` and `\)`, with the addition of:

1. A regular expression followed by + matches one or more occurrences of the regular expression.
2. A regular expression followed by ? matches 0 or 1 occurrences of the regular expression.
3. Two regular expressions separated by | or by a newline match strings that are matched by either.
4. A regular expression may be enclosed in parentheses () for grouping.

The order of precedence of operators is [], then *? +, then concatenation, then | and newline.

ERRORS

Exit status is 0 if any matches are found, 1 if none, 2 for syntax errors or inaccessible files (even if matches were found).

USAGE

General.

Lines are limited to BUFSIZ characters; longer lines are truncated. (BUFSIZ is defined in `/usr/include/stdio.h`.)

FUTURE DIRECTIONS

The functionality of `egrep` and `fgrep` will eventually be provided in GREP(BU_CMD), and these two commands discontinued.

SEE ALSO

ED(BU_CMD), GREP(BU_CMD), SED(BU_CMD).

LEVEL

Level 2: December 1, 1985.

NAME

ex — text editor

SYNOPSIS

```
ex [ - ] [ -v ] [ -r ] [ -R ] [ +command ] [ -l ]
[ file ... ]
```

DESCRIPTION

The command **ex** is a line oriented text editor, which supports both command and display editing [see VI(AU_CMD)]. The command line options are:

-	Suppress all interactive-user feedback. This is useful in processing editor scripts.
-v	Invokes **vi**
-r	Recover the named files after an editor or system crash. If no files are named a list of all saved files will be printed.
-R	*Readonly* mode set, prevents accidentally overwriting the file.
+command	Begin editing by executing the specified editor search or positioning **command**.
-l	LISP mode; indents appropriately for *lisp* code; the **( )** **{ }** **[[** and **]]** commands in **vi** are modified to have meaning for lisp.

The **file** argument(s) indicates files to be edited, in the order specified.

The name of the file being edited by **ex** is the **current** file. The text of the file is read into a buffer, and all editing changes are performed in this buffer; changes have no effect on the file until the buffer is written out explicitly.

The **alternate** file name is the name of the last file mentioned in an editor command, or the previous current file name if the last file mentioned became the current file. The character '%' in filenames is replaced by the current file name, and the character '#' by the alternate file name.

The named buffers **a** through **z** may be used for saving blocks of text during the edit. If the buffer name is specified in upper case, the buffer is appended to rather than being overwritten.

The read-only mode can be cleared from within the edit by setting the **noreadonly** edit option (see **Edit Options** below). Writing to a different file is allowed in read-only mode; in addition, the write can be forced by using '!' (see the **write** command below).

When an error occurs **ex** sends the BEL character to the terminal (to sound the bell) and prints a message. If an interrupt signal is received, **ex** returns to the command level in addition to the above actions. If the editor input is from a file, **ex** exits at the interrrupt. (The bell action may be disabled by

the use of an edit option, see below.)

If the system crashes, `ex` attempts to preserve the buffer if any unwrtten changes were made. The command line option `-r` is used to retrieve the saved changes.

At the beginning, `ex` is in the `command` mode, which is indicated by the ':' prompt. The `input` mode is entered by `append`, `insert`, or `change` commands; it is left (and command mode re-entered) by typing a period '.' alone at the beginning of a line.

Command lines beginning with the double quote character '"' are ignored. (this may used for comments in an editor script.)

Addressing

.	Dot '.' refers to the current line. There is always a current line; the positioning may be the result of an explicit movement by the user, or the result of a command that affected multiple lines (in which case it is usually the last line affected).
n	The nth line in the buffer, with lines numbered sequentially from 1.
$	The last line in the buffer.
%	Abbreviation for "1,$", the entire buffer.
+n -n	An offset relative to the current line. (The forms '.+3', '+3', and '+++' are equivalent.)
/pat/ ?pat?	Line containing the pattern (regular expression) `pat`, scanning forward (//) or backward (??). The trailing / or ? may be omitted if the line is only to be printed. If the pattern is omitted, the previous pattern specified is used.
`x	Lines may be marked using single lower case letters (see the `mark` command below); `x refers to line marked `x`. In addition, the previous current line is marked before each non-relative motion; this line may be referred to by using ` for `x`.

Addresses to commands consist of a series of line addresses (specified as above), separated by ',' or ';'. Such address lists are evaluated left-to-right. When ';' is the separator, the current line is set to the value of the previous address before the next address is interpreted. If more addresses are given than the command requires, then all but the last one or two are ignored. Where a command requires two addresses, the first line must precede the second one in the buffer. A null address in a

list defaults to the current line.

Command names and abbreviations

abbrev	**ab**	next	**n**	unmap	**unm**
append	**a**	number	**# nu**	version	**ve**
args	**ar**	preserve	**pre**	visual	**vi**
change	**c**	print	**p**	write	**w**
copy	**co**	put	**pu**	xit	**x**
delete	**d**	quit	**q**	yank	**ya**
edit	**e**	read	**re**	(window)	**z**
file	**f**	recover	**re**	(escape)	**!**
global	**g v**	rewind	**rew**	(lshift)	**<**
insert	**i**	set	**se**	(rshift)	**>**
join	**j**	shell	**sh**	(resubst)	**& s**
list	**l**	source	**so**	(scroll)	**^D**
map		substitute	**s**	(line no)	**=**
mark	**k ma**	unabbrev	**una**		
move	**m**	undo	**u**		

Command descriptions

In the following, `line` is a single line address, given in any of the forms described in the **Addressing** section above; `range` is a pair of line addresses, separated by a comma or semicolon (see the **Addressing** section for the difference between the two); `count` is a positive integer, specifying the number of lines to be affected by the command; `flags` is one or more of the characters '#', 'p', and 'l'; the corresponding command to print the line is executed after the command completes. Any number of '+' or '-' characters may also be given with these flags.

When `count` is used, `range` is not effective; only a line number should be specified instead, to indicate the first line affected by the command. (If a range is given, then the last line of the range is taken as the starting line for the command.)

These modifiers are all optional; the defaults are as follows, unless otherwise stated: the default for `line` is the current line; the default for `range` is the current line only (.,.); the default for `count` is 1; the default for `flags` is null.

When only a `line` or a `range` is specified (with a null command), the implied command is `print`; if a null line is entered, the next line is printed (equivalent to '`.+1p`')

`ab word rhs`
: Add the named abbreviation to the current list. In visual mode, if `word` is typed as a complete word during input, it is replaced by the string `rhs`.

`line a`
: Enters input mode; the input text is placed after the specified line.

If line 0 is specified, the text is placed at the beginning of the buffer. The last input line becomes the current line, or the target line, if no lines are input.

`ar`
The argument list is printed, with the current argument inside '[' and ']'.

`range c count`
Enters input mode; the input text replaces the specified lines. The last input line becomes the current line; if no lines are input, the effect is the same as a delete.

`range co line flags`
A copy of the specified lines (`range`) is placed after the specified destination line; line 0 specifes that the lines are to be placed at the beginning of the buffer.

`range d buffer count`
The specified lines are deleted from the buffer. If a named buffer is specified, the deleted text is saved in it. The line after the deleted lines becomes the current line, or the last line if the deleted lines were at the end.

`e +line file`
Begin editing a new file. If the current buffer has been modified since the last write, then a warning is printed and the command is aborted. This action may be overridden by appending the character '!' to the command ("`e! file`"). The current line is the last line of the buffer; however, if this command is executed from within `visual`, the current line is the first line of the buffer. If the `+line` option is specified, the current line is set to the specified position, where `line` may be a number (or `$`) or specified as "`/pat`" or "`?pat`".

`f`
Prints the current file name and other information, including the number of lines and the current position.

`range g /pat/ cmds`
First marks the lines within the given range that match the given pattern. Then the given command(s) are executed with '.' set to each marked line.

`Cmds` may be specified on multiple lines by hiding newlines with a backslash. If `cmds` are omitted, each line is printed. `Append`, `change`, and `insert` commands are omitted; the terminating dot may be omitted if it ends `cmds`. `Visual` commands are also permitted, and take input from the terminal.

The `global` command itself, and the `undo` command are not allowed in `cmds`. The edit options `autoprint`,

`autoindent` and `report` are inhibited.

`range v /pat/ cmds`
This is the same as the `global` command, except that `cmds` is run on the lines that **do not** match the pattern.

`line i`
Enters input mode; the input text is placed before the specified line. The last line input becomes the current line, or the line before the target line, if no lines were input.

`range j count flags`
Joins the text from the specified lines together into one line. White space is adjusted to provide at least one blank character, to if there was a period at the end of the line, or none if the first following character is a ')'. Extra white space at the start of a line is discarded.

Appending the command with a '!' causes a simpler join with no white space processing.

`range l count flags`
Prints the specified lines with tabs printed as '^I' and the end of each line marked with a trailing '$'. (The only useful flag is '#', for line numbers.) The last line printed becomes the current line.

`map x rhs`
The `map` command is used to define macros for use in `visual` mode. The first argument is a single character, or the sequence '#n', where `n` is a digit, to refer to the function key `n`. When this character or function key is typed in `visual` mode, the action is as if the corresponding `rhs` had been typed. If '!' is appended to the command `map`, then the mapping is effective during insert mode rather than command mode. Special characters, white space, and newline must be escaped with a control-V to be entered in the arguments.

`line ma x`
(The letter `k` is an alternative abbreviation for the `mark` command.) The specified line is given the specified mark `x`, which must be a single lower case letter. (The `x` must be preceded by a space or tab.) The current line position is not affected.

`range m line`
Moves the specified lines (`range`) to be after the target line. The first of the moved lines becomes the current line.

`n`
The next file from the command line argument list is edited. Appending a '!' to the command overrides the warning about the buffer having been modifed since the last write (discarding any changes). The argument list may be replaced by specifying a new

one on this command line.

`range nu count flags`
(The character '#' is an alternative abbreviation for the `number` command.) Prints the lines, each preceded by its line number. (The only useful flag is 'l'.) The last line printed becomes the current line.

`pre`
The current editor buffer is saved as though the system had just crashed. This command is for use in emergencies, for example when a write does not work, and the buffer cannot be saved in any other way.

`range p count`
Prints the specified lines, with non-printing characters printed as control characters in the form '^x'; DEL is represented as '^?'. The last line printed becomes the current line.

`line pu buffer`
Puts back deleted or "yanked" lines. A buffer may be specified; otherwise, the text in the unnamed buffer (where delted or yanked text is placed by default) is restored.

`q`
Causes termination of the edit. If the buffer has been modified since the last write, a warning is printed and the command fails. This warning may be overridden by appending a '!' to the command (discarding changes).

`line r file`
Places a copy of the specified file in the buffer after the target line (which may be line 0 to place text at the beginning). If no `file` is named the current file is the default. If there is no current file then `file` becomes the current file. The last line read becomes the current line; in `visual` the first line read becomes the current line.

If `file` is given as "`!string`" then `string` is taken to be a system command, and passed to the command interpreter; the resultant output is read in to the buffer. A blank or tab must precede the '!'.

`rec file`
Recovers `file` from the save area, after an accidental hangup or a system crash.

`rew`
The argument list is rewound, and the first file in the list is edited. Any warnings may be overridden by appending a '!'.

`se parameter`
With no arguments, the set command prints those options whose values have been changed from the default settings; with the parameter `all` it prints all of the option values.

Giving an option name followed by a '?' causes the current value of that option to be printed. The '?' is necessary only for Boolean valued options. Boolean options are given values by the form 'se `option`' to turn them on, or 'se `nooption`' to turn them off; string and numeric options are assigned by the form 'se `option`=value'. More than one parameter may be given; they are interpreted left to right.

See **Edit Options** below for further details about options.

`sh`
The user is put into the command interpreter [usually `sh`; see SH(BU_CMD)]; editing is resumed on exit.

`so file`
Reads and executes commands from the specified file. Such `so` commands may be nested.

`range s /pat/repl/ options count flags`
On each specified line, the first instance of the pattern `pat` is replaced by the string `repl`. (See **Regular Expressions** and **Replacement Strings** below.) If `options` includes the letter 'g' (global), then all instances of the pattern in the line are substituted. If the option letter 'c' (confirm) is included, then before each substitution the line is typed with the pattern to be replaced marked with '^' characters; a response of 'y' causes the substitution to be done, while any other input aborts it. The last line substituted becomes the current line.

`una word`
Delete `word` from the list of abbreviations.

`u`
Reverses the changes made by the previous editing command. For this purpose, `global` and `visual` are considered single commands. Commands which affect the external environment, such as `write`, `edit` and `next`, cannot be undone. An `undo` can itself be reversed.

`unm x`
The macro definition for `x` is removed.

`ve`
Prints the current version of the editor.

`line vi type count`
Enters visual mode at the specified line. The `type` is optional,

and may be '-' or '.', as in the `z` command, to specify the position of the specified line on the screen window. (The default is to place the line at the top of the screen window.) A `count` specifies an initial window size; the default is the value of the edit option `window`. The command `Q` exits visual mode. [For more information, see VI(AU_CMD)]

`range w file`
Writes the specified lines (the whole buffer, if no `range` is given) out to `file`, printing the number of lines and characters written. If `file` is not specified, the default is the current file. (The command fails with an error message if there is no current file and no file is specified.)

If an alternate file is specified, and the file exists, then the write will fail; it may be forced by appending a '!' to the command. An existing file may be appended to by appending '>>' to the command. If the file does not exist, an error is reported.

If the file is specified as '`!string`', then `string` is taken as a system command; the command interpreter is invoked, and the specified lines are passed as standard input to the command.

The command `wq` is equivalent to a `w` followed by a `q`; `wq!` is equivalent to `w!` followed by `q`.

`x`
Writes out the buffer if any changes have been made, and then (in any case) quits.

`range ya buffer count`
Places the specified lines in the named buffer. If no buffer is specified, the unnamed buffer is used (where the most recently deleted or yanked text is placed by default).

`line z type count`
If `type` is omitted, then `count` lines following the specified line (default current line) are printed. The default for `count` is the value of the edit option `window`.

If `type` is specified, it must be '-' or '.'; a '-' causes the line to be placed at the bottom of the screen, while a '.' causes the line to be placed in the middle. The last line printed becomes the current line.

`! command`
The remainder of the line after the '!' is passed to the system command interpreter for execution. A warning is issued if the buffer has been changed since the last write. A single '!' is printed when the command completes. The current line position is not affected.

Within the text of `command` '%' and '#' are expanded as filenames, and '!' is replaced with the text of the previous '!' command. (Thus '!!' repeats the previous '!' command.) If any such expansion is done, the expanded line will be echoed.

`range! command`
In this form of the '!' command, the specified lines (there is no default; see previous paragraph) are passed to the command interpreter as standard input; the resulting output replaces the specified lines.

`range < count`
Shift the specified lines to the left; the number of spaces to be shifted is determined by the edit option `shiftwidth`. Only white space (blanks and tabs)is lost in shifting; other characters are not affected. The last line changed becomes the current line.

`range > count`
Shift the specified lines to the right, by inserting white space (see previous paragraph for further details).

`range & options count flags`
Repeats the previous substitute command, as if '&' were replaced by the previous 's`/pat/repl/`'. (The same effect is obtained by omitting the '`/pat/repl/`' string in the `substitute` command.)

`^D` (control-D)
Control-D (ASCII EOT) prints the next `n` lines, where `n` is the value of the edit option `scroll`.

`line =`
Prints the line number of the specified line (default last line). The current line position is not affected.

Regular Expressions

Regular expressions are interpreted according to the setting of the edit option `magic`; the following assumes the setting `magic`. The differences caused by the setting `nomagic` are described below.

The following constructs are used to construct regular expressions:

`char` An ordinary character matches itself. The following characters are not ordinary, and must be escaped (preceded by '\') to have their ordinary meaning: '^' at the beginning of a pattern; '$' at the end of a pattern; '*' anywhere other than the beginning of a pattern; '.', '\', '[', and ' ', anywhere in a pattern.

When at the beginning of a pattern, matches the beginning of the line.

$ When at the end of a pattern, matches the end of the line.

. matches any single character in the line.

\< Matches the beginning of a "word". That is, the matched string must begin in a letter, digit, or underline, and be preceded by the beginning of the line or a character other than the above.

\> Matches the end of a "word" (see previous paragraph).

[string] Matches any single character in `string`. Within string, the following have special meanings: a pair of characters separated by '-' defines a range (e.g., '[a-z]' defines any lower case letter); the character '^', if it is the first one in `string`, causes the construct to match characters other than those specified in `string`. These special meaning can be removed by escaping the characters.

* Matches zero or more occurrences of the preceding regular expression.

Matches the replacement part of the last `substitute` command.

\(...\) A regular expression may be enclosed in escaped parentheses; this serves only to identify them for substitution actions.

A concatenation of two regular expressions is a regular expression that matches the concatenation of the strings matched by each component.

When `nomagic` is set, the only characters with special meanings are '^' at the beginning of a pattern, '$' at the end of a pattern, and '\'. The characters '.', '*', '[', and ' ' lose their special meanings, unless escaped by a '\'.

Replacement Strings

The character '&' ('\&' if `nomagic` is set) in the replacement string stands for the text matched by the pattern to be replaced. The character '~' ('\~' if `nomagic` is set) is replaced by the replacement part of the `previous` substitute command. The sequence '\n', where `n` is an integer, is replaced by the text matched by the pattern enclosed in the `n`th set of parentheses '\(' and '\)'. The sequence '\u' ('\l') causes the immediately following character in the replacement to be converted to upper-case (lower-case), if this character is a letter. The sequence '\U' ('\L') turns such conversion on, until the sequence '\E' or '\e' is encountered, or the end of the replacement string is reached.

Edit Options

The command `ex` has a number of options that modify its behavior. These options have default settings, which may be changed using the `set` command (see above). Options may also be set at startup by

putting a `set` command string in the environmental variable `EXINIT`, or in the file `.exrc` in the `HOME` directory, or in `.exrc` in the current directory.

Options are Boolean unless otherwise specified.

`autoindent, ai`
: If autoindent is set, each line in insert mode is indented (using blanks and tabs) to align with the previous line. (Starting indentation is determined by the line appended after, or the line inserted before, or the first line changed.) Additional indentation can be provided as usual; succeeding lines will automatically be indented to the new alignment. Reducing the indent is achieved by typing control-D one or more times; the cursor is moved back `shiftwidth` spaces for each control-D. (A '^' followed by a control-D removes all indentation temporarily for the current line; a '0' followed by a control-D removes all indentation.)

`autoprint, ap`
: The current line is printed after each command that changes buffer text. (Autoprint is suppressed in globals.)

`autowrite, aw`
: The buffer is written (to the current file) if it has been modified, and a `next`, `rewind`, or `!` command is given.

`beautify, bf`
: Causes all control characters other than tab, newline and formfeed to be discarded from the input text.

`directory, dir`
: The value of this option specifies the directory in which the editor buffer is to be placed. If this directory is not writeable by the user, the editor quits.

`edcompatible, ed`
: Causes the presence of `g` and `c` suffixes on substitute commands to be remembered, and toggled by repeating the suffixes.

`ignorecase, ic`
: All upper case characters in the text are mapped to lower case in regular expression matching. Also, all upper case characters in regular expressions are mapped to lower case, except in character class specifications.

`lisp`
: `Autoindent` mode, and the `( ) { } [[ ]]` commands in `visual` are suitable modified for *lisp* code.

`list`
: All printed lines will be displayed with tabs shown as '^I', and the end of line marked by a '$'.

`magic`
Changes interpretation of characters in regular expressions and substitution replacement strings (see the relevant sections above).

`number, nu`
Causes lines to be printed with line numbers.

`paragraphs, para`
The value of this option is a string, in which successive pairs of characters specify the names of text-processing macros which begin paragraphs. (A macro appears in the text in the form '`.XX`', where the '.' is the first character in the line.)

`prompt`
When set, command mode input is prompted for with a ':'; when unset, no prompt is displayed.

`redraw`
The editor simulates an intelligent terminal on a dumb terminal. (Since this is likely to require a large amount of output to the terminal, it is useful only at high transmission speeds.)

`remap`
If set, then macro translation allows for macros defined in terms of other macros; translation continues until the final product is obtained. If unset, then a one-step translation only is done.

`report`
The value of this option gives the number of lines that must be changed by a command before a report is generated on the number of lines affected.

`scroll`
The value of this option determines the number of lines scrolled on a control-D, and the number of lines displayed by the `z` command (twice the value of scroll).

`sections`
The value of this option is a string, in which successive pairs of characters specify the names of text-processing macros which begin sections. (See `paragraphs` option above.)

`shiftwidth, sw`
The value of this option gives the width of a software tab stop, used during `autoindent`, and by the shift commands.

`showmatch, sm`
In `visual` mode, when a) or } is typed, the matching (or { is shown if it is still on the screen.

`slowopen, slow`
In `visual` mode, prevents screen updates during input to improve throughput on unintelligent terminals.

`tabstop, ts`
The value of this options specifies the software tab stops to be used by the editor to expand tabs in the input file.

`terse`
When set, error messages are shorter.

`window`
The number of lines in a text window in `visual` mode.

`wrapscan, ws`
When set, searches (using '//' or '??') wrap around the end of the file; when unset, searches stop at the beginning or the end of the file, as appropriate.

`wrapmargin, wm`
In `visual` mode, if the value of this option is greater than zero (say `n`), then a newline is automatically added to an input line, at a word boundary, so that lines end at least `n` spaces from the right margin of the terminal screen.

`writeany, wa`
Inhibits the checks otherwise made before write commands, allowing a write to any file (provided the system allows it).

FILES

`/usr/lib/terminfo/*/*`	`terminfo` terminal capability database
`$HOME/.exrc`	editor initialization file
`./.exrc`	editor initialization file

USAGE

End-user.

The `undo` command causes all marks to be lost on lines that were changed and then restored.

The `z` command prints a number of logical rather than physical lines. More than a screen-ful of output may result if long lines are present.

Null characters are discarded in input files and cannot appear in resultant files.

SEE ALSO

VI(AU_CMD).

LEVEL

Level 1.

NAME

id — print user and group IDs and names

SYNOPSIS

`id`

DESCRIPTION

The command `id` writes a message on the standard output giving the user and group IDs and the corresponding names of the invoking process. If the effective and real IDs do not match, both are printed.

USAGE

General.

SEE ALSO

LOGNAME(AU_CMD), GETUID(BA_OS).

LEVEL

Level 1.

NAME

join — join two files on identical-valued field

SYNOPSIS

`join [ options ] file1 file2`

DESCRIPTION

The command `join` performs an "equality join" on the files `file1` and `file2`. If `file1` is `-`, the standard input is used in its place.

A field must be specified for each file as the "join field", on which the files are compared. There is one line in the output for each pair of lines in `file1` and `file2` that have identical join fields. The output line normally consists of the common field, then the rest of the line from `file1`, then the rest of the line from `file2`. This format can be changed by using the `-o` option (see below).

The files `file1` and `file2` must be sorted in increasing ASCII collating sequence on the fields on which they are to be joined, normally the first in each line.

The default input field separators are blank, tab, or newline. In this case, multiple separators count as one field separator, and leading separators are ignored. The default output field separator is a blank.

Some of the options below use the argument `n`. This argument should be a `1` or a `2` referring to either `file1` or `file2`, respectively. The following options are recognized:

`-an` In addition to the normal output, produce a line for each unpairable line in file `n`, where `n` is 1 or 2.

`-es` Replace empty output fields by string `s`.

`-jnm` Join on the `m`th field of file `n`. If `n` is missing, use the `m`th field in each file. Fields are numbered starting with `1`.

`-olist` Each output line comprises the fields specified in `list`, each element of which has the form `n.m`,1 where `n` is a file number and `m` is a field number. The common field is not printed unless specifically requested.

`-tc` Use character `c` as a separator, for both input and output. Every appearance of `c` in a line is significant.

USAGE

General.

Filenames that are numeric may cause conflict when the `-o` option is used right before listing filenames.

SEE ALSO

AWK(BU_CMD), COMM(BU_CMD), SORT(BU_CMD), UNIQ(BU_CMD).

LEVEL

Level 1.

NAME

logname — get login name

SYNOPSIS

`logname`

DESCRIPTION

The command `logname` returns the user's login name.

USAGE

General.

LEVEL

Level 1.

NAME

lp, cancel — send/cancel requests to an LP line printer

SYNOPSIS

```
lp [-c] [-ddest] [-m] [-nnumber] [-ooption] [-s]
[-ttitle] [-w] files

cancel [ids] [printers]
```

DESCRIPTION

The command `lp` arranges for the named files and associated information (collectively called a `request`) to be printed by a line printer. If no file names are mentioned, the standard input is assumed. The file name `-` stands for the standard input and may be supplied on the command line in conjunction with named `files`. The order in which `files` appear is the same order in which they will be printed.

`lp` associates a unique ID with each request and prints it on the standard output. This ID can be used later to cancel (see `cancel`) or find the status [see LPSTAT(AU_CMD)] of the request.

The following options to `lp` may appear in any order and may be intermixed with file names:

`-c` Make copies of the `files` to be printed immediately when `lp` is invoked. Normally, `files` will not be copied, but will be linked whenever possible. If the `-c` option is not given, then the user should be careful not to remove any of the `files` before the request has been printed in its entirety. It should also be noted that in the absence of the `-c` option, any changes made to the named `files` after the request is made but before it is printed will be reflected in the printed output.

`-ddest` Choose `dest` as the printer or class of printers that is to do the printing. If `dest` is a printer, then the request will be printed only on that specific printer. If `dest` is a class of printers, then the request will be printed on the first available printer that is a member of the class. Under certain conditions (printer unavailability, file space limitation, etc.), requests for specific destinations may not be accepted [see LPSTAT(AU_CMD)]. By default, `dest` is taken from the environmental variable `LPDEST` (if it is set). Otherwise, a default destination (if one exists) for the computer system is used. Destination names vary between systems [see LPSTAT(AU_CMD)].

`-m` Send mail [see MAIL(BU_CMD)] after the files have been printed. By default, no mail is sent upon normal completion of the print request.

`-nnumber` Print `number` copies (default of 1) of the output.

`-ooption` Specify printer-dependent or class-dependent `options`. Several such `options` may be collected by specifying the `-o` keyletter more than once.

`-s` Suppress messages from `lp` such as "request id is ...".

`-ttitle` Print `title` on the banner page of the output.

`-w` Write a message on the user's terminal after the `files` have been printed. If the user is not logged in, then mail will be sent instead.

The command `cancel` cancels line printer requests that were made by the `lp` command. The command line arguments may be either request `ids` (as returned by `lp`) or `printer` names [for a complete list, use LPSTAT(AU_CMD).] Specifying a request `id` cancels the associated request even if it is currently printing. Specifying a `printer` cancels the request which is currently printing on that printer. In either case, the cancellation of a request that is currently printing frees the printer to print its next available request.

USAGE

General.

SEE ALSO

LPSTAT(AU_CMD), MAIL(BU_CMD),

LEVEL

Level 1.

NAME

lpstat – print LP status information

SYNOPSIS

`lpstat [options]`

DESCRIPTION

The command `lpstat` prints information about the current status of the LP line printer system.

If no `options` are given, then `lpstat` prints the status of all requests made to LP(AU_CMD) by the user. Any arguments that are not `options` are assumed to be request `ids` as returned by `lp` [see LP(AU_CMD)]. The command `lpstat` prints the status of such requests. The `options` may appear in any order and may be repeated and intermixed with other arguments. Some of the keyletters below may be followed by an optional `list` that can be in one of two forms: a list of items separated from one another by a comma, or a list of items enclosed in double quotes and separated from one another by a comma and/or one or more spaces. For example:

–u"user1, user2, user3"

The omission of a `list` following such keyletters causes all information relevant to the keyletter to be printed, for example:

lpstat –o

prints the status of all output requests.

–a[`list`] Print acceptance status of destinations for output requests. `list` is a list of intermixed printer names and class names.

–c[`list`] Print class names and their members. `list` is a list of class names.

–d Print the system default destination for output requests.

–o[`list`] Print the status of output requests. `list` is a list of intermixed printer names, class names, and request `ids`.

–p[`list`] Print the status of printers. `list` is a list of printer names.

–r Print the status of the LP request scheduler

–s Print a status summary, including the status of the line printer scheduler, the system default destination, a list of class names and their members, and a list of printers and their associated devices.

–t Print all status information.

–u[`list`] Print status of output requests for users. `list` is a list of login names.

–v[`list`] Print the names of printers and the path names of the devices associated with them. `list` is a list of printer names.

USAGE

General.

SEE ALSO

LP(AU_CMD).

LEVEL

Level 1.

NAME

mailx — interactive message processing system

SYNOPSIS

`mailx [options] [name...]`

DESCRIPTION

The command `mailx` provides a comfortable, flexible environment for sending and receiving messages electronically. When reading mail, `mailx` provides commands to facilitate saving, deleting, and responding to messages. When sending mail, `mailx` allows editing, reviewing and other modification of the message as it is entered.

Incoming mail is stored in a standard file for each user, called the system *mailbox* for that user. When `mailx` is called to read messages, the *mailbox* is the default place to find them. As messages are read, they are marked to be moved to a secondary file for storage, unless specific action is taken, so that the messages need not be seen again. This secondary file is called the *mbox* and is normally located in the user's HOME directory (see `MBOX`, in **ENVIRONMENT VARIABLES** below for a description of this file). Messages remain in this file until specifically removed.

On the command line, `options` start with a dash (–) and any other arguments are taken to be destinations (recipients). If no recipients are specified, `mailx` will attempt to read messages from the *mailbox*. Command line options are:

`–e` Test for presence of mail. The command `mailx` prints nothing and exits with a successful return code if there is mail to read.

`–f` [*filename*] Read messages from *filename* instead of *mailbox*. If no *filename* is specified, the *mbox* is used.

`–F` Record the message in a file named after the first recipient. Overrides the "record" variable, if set (see **ENVIRONMENT VARIABLES**).

`–h`*number* The number of network "hops" made so far. This is provided for network software to avoid infinite delivery loops.

`–H` Print header summary only.

`–i` Ignore interrupts. See also "ignore" (**ENVIRONMENT VARIABLES**).

`–n` Do not initialize from the system default *Mailx.rc* file.

`–N` Do not print initial header summary.

`–r`*address* Pass *address* to network delivery software. All tilde commands are disabled.

−s*subject* Set the Subject header field to *subject*.

−u*user* Read *user*'s *mailbox*. This is only effective if *user*'s *mailbox* is not read protected.

When reading mail, `mailx` is in *command mode*. A header summary of the first several messages is displayed, followed by a prompt indicating `mailx` can accept regular commands (see COMMANDS below). When sending mail, `mailx` is in *input mode*. If no subject is specified on the command line, a prompt for the subject is printed. As the message is typed, `mailx` will read the message and store it in a temporary file. Commands may be entered by beginning a line with the tilde (~) escape character followed by a single command letter and optional arguments. See **TILDE ESCAPES** for a summary of these commands.

At any time, the behavior of `mailx` is governed by a set of *environmental variables*. These are flags and valued parameters which are set and cleared via the `set` and `unset` commands. See **ENVIRONMENT VARIABLES** below for a summary of these parameters.

Regular commands are of the form

```
[ command ][ msglist ][ arguments ]
```

If no command is specified in *command mode*, **print** is assumed. In *input mode*, commands are recognized by the escape character, and lines not treated as commands are taken as input for the message.

Each message is assigned a sequential number, and there is at any time the notion of a 'current' message, marked by a '>' in the header summary. Many commands take an optional list of messages (*msglist*) to operate on, which defaults to the current message. A *msglist* is a list of message specifications separated by spaces, which may include:

n	Message number **n**.
.	The current message.
^	The first undeleted message.
$	The last message.
*	All messages.
n−m	An inclusive range of message numbers.
user	All messages from **user**.
/string	All messages with **string** in the subject line (case ignored).
:c	All messages of type c, where c is one of:
d	deleted messages
n	new messages
o	old messages
r	read messages
u	unread messages

Note that the context of the command determines whether this type of message specification makes sense.

Other arguments are usually arbitrary strings whose usage depends on the command involved. File names, where expected, can be specified with meta-characters understood by the command interpreter. Special characters are recognized by certain commands and are documented with the commands below.

At start-up time, `mailx` reads commands from a system-wide file to initialize certain parameters, then from a private start-up file (`$HOME/.mailrc`) for personalized variables. Most regular commands are legal inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are not legal in the start-up file: `!`, `Copy`, `edit`, `followup`, `Followup`, `hold`, `mail`, `preserve`, `reply`, `Reply`, `shell`, and `visual`. Any errors in the start-up file cause the remaining lines in the file to be ignored.

COMMANDS

The following is a complete list of `mailx` commands:

`!` *command*
Escape to the command interpreter. See "`SHELL`" (**ENVIRONMENT VARIABLES**).

`# comment`
Null command (comment). This may be useful in *.mailrc* files.

`=`
Print the current message number.

`?`
Prints a summary of commands.

`alias` *alias name ...*

`group` *alias name ...*
Declare an alias for the given names. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

`alternates` *name ...*
Declares a list of alternate names for the user's login. When responding to a message, these names are removed from the list of recipients for the response. With no arguments, **alternates** prints the current list of alternate names. See also "allnet" (**ENVIRONMENT VARIABLES**).

`cd` [*directory*]

`chdir` [*directory*]
Change directory. If *directory* is not specified, `$HOME` is used.

`copy` [*filename*]

`copy` [*msglist*] *filename*
Copy messages to the file without marking the messages as saved. Otherwise equivalent to the `save` command.

`Copy` [*msglist*]
Save the specified messages in a file whose name is derived from the author of the message to be saved, without marking the messages as saved. Otherwise equivalent to the `Save` command.

`delete` [*msglist*]
Delete messages from the *mailbox*. If "autoprint" is set, the next message after the last one deleted is printed (see **ENVIRONMENT VARIABLES**).

`discard` [*header-field* ...]
`ignore` [*header-field* ...]
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." The fields are included when the message is saved. The `Print` and `Type` commands override this command.

`dp` [*msglist*]
`dt` [*msglist*]
Delete the specified messages from the *mailbox* and print the next message after the last one deleted. Roughly equivalent to a `delete` command followed by a `print` command.

`echo` *string* ...
Echo the given strings (like ECHO(BU_CMD).)

`edit` [*msglist*]
Edit the given messages. The messages are placed in a temporary file and the "`EDITOR`" variable is used to get the name of the editor (see **ENVIRONMENT VARIABLES**). Default editor is `ed`.

`exit`
`xit`
Exit from `mailx`, without changing the *mailbox*. No messages are saved in the *mbox* (see also `quit`).

`file` [*filename*]
`folder` [*filename*]
Quit from the current file of messages and read in the specified file. Several special characters are recognized when used as file names, with the following substitutions:

%	the current *mailbox*.
%user	the *mailbox* for user.
#	the previous file.
&	the current *mbox*.

Default file is the current *mailbox*.

`folders`
Print the names of the files in the directory set by the "folder" variable (see **ENVIRONMENT VARIABLES**).

`followup` [*message*]
Respond to a message, recording the response in a file whose name is derived from the author of the message. Overrides the "record" variable, if set. See also the `Followup`, `Save`, and `Copy` commands and "outfolder" (**ENVIRONMENT VARIABLES**).

`Followup` [*msglist*]
Respond to the first message in the *msglist*, sending the message to the author of each message in the *msglist*. The subject line is taken from the first message and the response is recorded in a file whose name is derived from the author of the first message. See also the `followup`, `Save`, and `Copy` commands and "outfolder" (**ENVIRONMENT VARIABLES**).

`from` [*msglist*]
Prints the header summary for the specified messages.

`group` *alias name ...*
`alias` *alias name ...*
Declare an alias for the given *names*. The names will be substituted when *alias* is used as a recipient. Useful in the *.mailrc* file.

`headers` [*message*]
Prints the page of headers which includes the message specified. The "screen" variable sets the number of headers per page (see **ENVIRONMENT VARIABLES**). See also the `z` command.

`help`
Prints a summary of commands.

`hold` [*msglist*]
`preserve` [*msglist*]
Holds the specified messages in the *mailbox*.

`if s|r`
mail-commands
`else` *mail-commands*
`endif`
Conditional execution, where `s` will execute following *mail-commands*, up to an `else` or `endif`, if the program is in *send* mode, and `r` causes the *mail-commands* to be executed only in *receive* mode. Useful in the *.mailrc* file.

`ignore` *header-field ...*
`discard` *header-field ...*
Suppresses printing of the specified header fields when displaying messages on the screen. Examples of header fields to ignore are "status" and "cc." All fields are included when the message is saved. The `Print` and `Type` commands override this command.

`list`
Prints all commands available. No explanation is given.

mail *name* ...
Mail a message to the specified users.

mbox [*msglist*]
Arrange for the given messages to end up in the standard *mbox* save file when `mailx` terminates normally. See `MBOX` (**ENVIRONMENT VARIABLES**) for a description of this file. See also the `exit` and `quit` commands.

next [*message*]
Go to next message matching *message*. A *msglist* may be specified, but in this case the first valid message in the list is the only one used. This is useful for jumping to the next message from a specific user, since the name would be taken as a command in the absence of a real command. See the discussion of *msglists* above for a description of possible message specifications.

pipe [*msglist*] [*command*]
| [*msglist*] [*command*]
Pipe the message through the given *command*. The message is treated as if it were read. If no arguments are given, the current message is piped through the command specified by the value of the "cmd" variable. If the "page" variable is set, a form feed character is inserted after each message (see **ENVIRONMENT VARIABLES**).

preserve [*msglist*]
hold [*msglist*]
Preserve the specified messages in the *mailbox*.

Print [*msglist*]
Type [*msglist*]
Print the specified messages on the screen, including all header fields. Overrides suppression of fields by the `ignore` command.

print [*msglist*]
type [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the `PAGER` environment variable. The default command is `pg`. (See **ENVIRONMENT VARIABLES**).

quit
Exit from `mailx`, storing messages that were read in *mbox* and unread messages in the *mailbox*. Messages that have been explicitly saved in a file are deleted.

Reply [*msglist*]
Respond [*msglist*]
Send a response to the author of each message in the *msglist*. The subject line is taken from the first message. If "record" is set to a file name, the response is saved at the end of that file (see **ENVIRONMENT VARIABLES**).

reply [*message*]
respond [*message*]
Reply to the specified message, including all other recipients of the message. If "record" is set to a file name, the response is saved at the end of that file (see **ENVIRONMENT VARIABLES**).

Save [*msglist*]
Save the specified messages in a file whose name is derived from the author of the first message. The name of the file is taken to be the author's name with all network addressing stripped off. See also the Copy, followup, and Followup commands and "outfolder" (**ENVIRONMENT VARIABLES**).

save [*filename*]
save [*msglist*] *filename*
Save the specified messages in the given file. The file is created if it does not exist. The message is deleted from the *mailbox* when mailx terminates unless "keepsave" is set (see also **ENVIRONMENT VARIABLES** and the exit and quit commands).

set
set *name*
set *name*=*string*
set *name*=*number*
Define a variable called *name*. The variable may be given a null, string, or numeric value. set by itself prints all defined variables and their values. See **ENVIRONMENT VARIABLES** for detailed descriptions of the mailx variables.

shell
Invoke an interactive command interpreter (see also SHELL (**ENVIRONMENT VARIABLES**)).

size [*msglist*]
Print the size in characters of the specified messages.

source *filename*
Read commands from the given file and return to command mode.

top [*msglist*]
Print the top few lines of the specified messages. If the "toplines" variable is set, it is taken as the number of lines to print (see **ENVIRONMENT VARIABLES**). The default is 5.

touch [*msglist*]
Touch the specified messages. If any message in *msglist* is not specifically saved in a file, it will be placed in the *mbox* upon normal termination. See exit and quit.

Type [*msglist*]
Print [*msglist*]
Print the specified messages on the screen, including all header fields.

Overrides suppression of fields by the `ignore` command.

`type` [*msglist*]
`print` [*msglist*]
Print the specified messages. If "crt" is set, the messages longer than the number of lines specified by the "crt" variable are paged through the command specified by the `PAGER` variable. The default command is `pg`. (See **ENVIRONMENT VARIABLES**).

`undelete` [*msglist*]
Restore the specified deleted messages. Will only restore messages deleted in the current mail session. If "autoprint" is set, the last message of those restored is printed (see **ENVIRONMENT VARIABLES**).

`unset` *name ...*
Causes the specified variables to be erased. If the variable was imported from the execution environment (i.e., an environment variable) then it cannot be erased.

`version`
Prints the current version and release date.

`visual` [*msglist*]
Edit the given messages with a screen editor. The messages are placed in a temporary file and the `VISUAL` variable is used to get the name of the editor (see **ENVIRONMENT VARIABLES**).

`write` [*msglist*] *filename*
Write the given messages on the specified file, minus the header and trailing blank line. Otherwise equivalent to the `save` command.

`xit`
`exit`
Exit from `mailx`, without changing the *mailbox*. No messages are saved in the *mbox* (see also `quit`).

`z`[+|-]
Scroll the header display forward or backward one screen—full. The number of headers displayed is set by the "screen" variable (see **ENVIRONMENT VARIABLES**).

TILDE ESCAPES

The following commands may be entered only from *input mode*, by beginning a line with the tilde escape character (~). See "escape" (**ENVIRONMENT VARIABLES**) for changing this special character.

~ ! *command*
Escape to the command interpreter.

~ . Simulate end of file (terminate message input).

~ : *mail-command*
~ _ *mail-command*

Perform the command-level request. Valid only when sending a message while reading mail.

`~?` Print a summary of tilde escapes.

`~A` Insert the autograph string "Sign" into the message (see **ENVIRONMENT VARIABLES**).

`~a` Insert the autograph string "sign" into the message (see **ENVIRONMENT VARIABLES**).

`~b` *name ...*
Add the *name*s to the blind carbon copy (Bcc) list.

`~c` *name ...*
Add the *name*s to the carbon copy (Cc) list.

`~d` Read in the *dead.letter* file. See "DEAD" (**ENVIRONMENT VARIABLES**) for a description of this file.

`~e` Invoke the editor on the partial message. See also `EDITOR` (**ENVIRONMENT VARIABLES**).

`~f` [*msglist*]
Forward the specified messages. The messages are inserted into the message, without alteration.

`~h` Prompt for Subject line and To, Cc, and Bcc lists. If the field is displayed with an initial value, it may be edited as if it had just been typed.

`~i` *string*
Insert the value of the named variable into the text of the message. For example, `~A` is equivalent to

`~m` [*msglist*]
Insert the specified messages into the letter, shifting the new text to the right one tab stop. Valid only when sending a message while reading mail.

`~p` Print the message being entered.

`~q` Quit from input mode by simulating an interrupt. If the body of the message is not null, the partial message is saved in *dead.letter*. See `DEAD` (**ENVIRONMENT VARIABLES**) for a description of this file.

`~r` *filename*
`~<`*filename*
`~<!`*command*
Read in the specified file. If the argument begins with an exclamation point (!), the rest of the string is taken as an arbitrary system command and is executed, with the standard output inserted into the message.

~s *string ...*
Set the subject line to *string.*

~t *name ...*
Add the given *names* to the To list.

~v Invoke a preferred screen editor on the partial message. See also VISUAL" (**ENVIRONMENT VARIABLES**).

~w *filename*
Write the partial message onto the given file, without the header.

~x Exit as with ~q except the message is not saved in *dead.letter.*

~ *command*
Pipe the body of the message through the given *command.* If the *command* returns a successful exit status, the output of the command replaces the message.

ENVIRONMENT VARIABLES

The following are environment variables taken from the execution environment and are not alterable within mailx.

HOME=*directory*
The user's base of operations.

MAILRC=*filename*
The name of the start-up file. Default is $HOME/.mailrc.

The following variables are internal mailx variables. They may be imported from the execution environment or set via the set command at any time. The unset command may be used to erase variables.

allnet All network names whose last component (login name) match are treated as identical. This causes the *msglist* message specifications to behave similarly. Default is noallnet. See also the alternates command and the "metoo" variable.

append
Upon termination, append messages to the end of the *mbox* file instead of prepending them. Default is noappend.

askcc
Prompt for the Cc list after message is entered. Default is noaskcc.

asksub
Prompt for subject if it is not specified on the command line with the -s option. Enabled by default.

autoprint
Enable automatic printing of messages after delete and undelete commands. Default is noautoprint.

bang
Enable the special-case treatment of exclamation points (!) in escape

command lines as in VI(AU_CMD). Default is `nobang`.

`cmd=`*command*
Set the default command for the `pipe` command. No default value.

`conv=conversion`
Convert uucp addresses to the specified address style. Conversion is disabled by default. See also "sendmail" and the `-U` command line option.

`crt=`*number*
Pipe messages having more than `number` lines through the command specified by the value of the "PAGER" variable (PG(BU_CMD) by default). Disabled by default.

`DEAD=`*filename*
The name of the file in which to save partial letters in case of untimely interrupt or delivery errors. Default is $HOME/dead.letter.

`debug`
Enable verbose diagnostics for debugging. Messages are not delivered. Default is `nodebug`.

`dot`
Take a period on a line by itself during input from a terminal as end-of-file. Default is `nodot`.

`EDITOR=`*command*
The command to run when the `edit` or `~e` command is used. Default is ED(BU_CMD).

`escape=c`
Substitute `c` for the ~ escape character.

`folder=`*directory*
The directory for saving standard mail files. User-specified file names beginning with a plus (+) are expanded by preceding the file name with this directory name to obtain the real file name. If *directory* does not start with a slash (/), $HOME is prepended to it. In order to use the plus (+) construct on a `mailx` command line, "folder" must be an exported environment variable. There is no default for the "folder" variable. See also "outfolder" below.

`header`
Enable printing of the header summary when entering `mailx`. Enabled by default.

`hold`
Preserve all messages that are read in the *mailbox* instead of putting them in the standard *mbox* save file. Default is `nohold`.

`ignore`
Ignore interrupts while entering messages. Handy for noisy dial-up

lines. Default is `noignore`.

`ignoreeof`
Ignore end-of-file during message input. Input must be terminated by a period (.) on a line by itself or by the ~. command. Default is `noignoreeof`. See also "dot" above.

`keep`
When the *mailbox* is empty, truncate it to zero length instead of removing it. Disabled by default.

`keepsave`
Keep messages that have been saved in other files in the *mailbox* instead of deleting them. Default is `nokeepsave`.

`MBOX`=*filename*
The name of the file to save messages which have been read. The `xit` command overrides this function, as does saving the message explicitly in another file. Default is `$HOME/mbox`.

`metoo`
If the user's login appears as a recipient, do not delete it from the list. Default is `nometoo`.

`LISTER`=*command*
The command (and options) to use when listing the contents of the "folder" directory. The default is `ls`.

`onehop`
When responding to a message that was originally sent to several recipients, the other recipient addresses are normally forced to be relative to the originating author's machine for the response. This flag disables alteration of the recipients' addresses, improving efficiency in a network where all machines can send directly to all other machines (i.e., one hop away).

`outfolder`
Causes the files used to record outgoing messages to be located in the directory specified by the "folder" variable unless the path name is absolute. Default is `nooutfolder`. See "folder" above and the `Save`, `Copy`, `followup`, and `Followup` commands.

`page`
Used with the `pipe` command to insert a form feed after each message sent through the pipe. Default is `nopage`.

`PAGER`=*command*
The command to use as a filter for paginating output. This can also be used to specify the options to be used. Default is `pg`.

`prompt`=*string*
Set the `command mode` prompt to *string*. Default is "? ".

`quiet`
Refrain from printing the opening message and version when entering `mailx`. Default is `noquiet`.

`record=`*filename*
Record all outgoing mail in *filename*. Disabled by default. See also "outfolder" above.

`save`
Enable saving of messages in *dead.letter* on interrupt or delivery error. See "DEAD" for a description of this file. Enabled by default.

`screen=`*number*
Sets the number of lines in a screen–full of headers for the `headers` command.

`sendmail=`*command*
Alternate command for delivering messages. Default is `mail`.

`sendwait`
Wait for background mailer to finish before returning. Default is `nosendwait`.

`SHELL=`*command*
The name of a preferred command interpreter. Default is `sh`.

`showto`
When displaying the header summary and the message is from the user, print the recipient's name instead of the author's name.

`sign=`*string*
The variable inserted into the text of a message when the `~a` (autograph) command is given. No default (see also `~i` (`TILDE ESCAPES`)).

`Sign=`*string*
The variable inserted into the text of a message when the `~A` command is given. No default (see also `~i` (`TILDE ESCAPES`)).

`toplines=`*number*
The number of lines of header to print with the `top` command. Default is 5.

`VISUAL=`*command*
The name of a preferred screen editor. Default is `vi`.

FILES

$HOME/.mailrc	users's start-up file
$HOME/mbox	secondary storage file

USAGE
End-user.

SEE ALSO

MAIL(BU_CMD), PG(BU_CMD), LS(BU_CMD), VI(AU_CMD).

LEVEL

Level 1.

New in System V Release 2.

NAME

mesg — permit or deny messages

SYNOPSIS

`mesg [ y | n ]`

DESCRIPTION

The command `mesg` with argument `n` prevents another user from writing to the invoking user's terminal, (e.g., by using `write` [see WRITE(AU_CMD)]). The command `mesg` with argument `y` reinstates write permission. With no arguments, `mesg` reports the current state without changing it.

ERRORS

Exit status is 0 if messages are receivable, 1 if not, 2 on error.

FILES

`/dev/tty*`

USAGE

General.

SEE ALSO

WRITE(AU_CMD).

LEVEL

Level 1.

NAME

newgrp — change to a new group

SYNOPSIS

`newgrp [-] [ group ]`

DESCRIPTION

The command `newgrp` changes a user's group identification. The user remains logged in and the current directory is unchanged, but calculations of access permissions to files are performed with respect to the new real and effective group IDs.

Exported environment variables retain their values after invoking `newgrp`; however, all unexported variables are either reset to their default value or set to null. Environment variables (such as `PS1`, `PS2`, `PATH`, `MAIL`, and `HOME`), unless exported, are reset to default values.

With no arguments, `newgrp` changes the group identification back to the group specified in the user's password file entry.

If the first argument to `newgrp` is a `-`, the environment is changed to what would be expected if the user actually logged in again.

FILES

`/etc/group`	system's group file
`/etc/passwd`	system's password file

USAGE

End-user.

SEE ALSO

SH(BU_CMD).

LEVEL

Level 1.

NAME

news — print news items

SYNOPSIS

```
news [ -a ] [ -n ] [ -s ] [ items ]
```

DESCRIPTION

The command `news` prints files from the system news directory.

When invoked without arguments, `news` prints the contents of all current files in the news directory, most recent first, with each preceded by an appropriate header. `news` stores the "currency" time as the modification date of a file named `.news_time` in the user's home directory (the identity of this directory is determined by the environmental variable `HOME`); only files more recent than this currency time are considered "current."

The `-a` option causes `news` to print all items, regardless of currency. In this case, the stored time is not changed.

The `-n` option causes `news` to report the names of the current items without printing their contents, and without changing the stored time.

The `-s` option causes `news` to report how many current items exist, without printing their names or contents, and without changing the stored time.

All other arguments are assumed to be specific news items that are to be printed.

If an interrupt (DEL or BREAK) is typed during the printing of a news item, printing stops and the next item is started. Another `interrupt` within one second of the first causes the program to terminate.

FILES

```
/etc/profile
$HOME/.news_time
```

USAGE

End-user.

LEVEL

Level 1.

NAME

od — octal dump

SYNOPSIS

```
od [ -bcdosx ] [ file ] [ [+]offset[.][b] ]
```

DESCRIPTION

The command `od` prints `file` in one or more formats as selected by the options. If no file is specified, the standard input is used. If no option is specified, `-o` is the default.

For the purposes of this description, *word* refers to a 16-bit unit, independent of the word size of the machine.

The meanings of the options are:

`-b` Interpret bytes in octal.

`-c` Interpret bytes in ASCII. Certain non-graphic characters appear as C escapes: NUL=`\0`, BS=`\b`, FF=`\f`, NL=`\n`, CR=`\r`, HT=`\t`; others appear as 3-digit octal numbers.

`-d` Interpret *words* in unsigned decimal.

`-o` Interpret *words* in octal.

`-s` Interpret *words* in signed decimal.

`-x` Interpret *words* in hex.

The offset argument specifies the offset in the file where dumping is to commence. This argument is normally interpreted as octal bytes. If `.` is appended, the offset is interpreted in decimal. If `b` is appended, the offset is interpreted in units of 512 bytes. If the file argument is omitted, the offset argument must be preceded by `+`.

USAGE

General.

LEVEL

Level 1.

NAME

passwd — change login password

SYNOPSIS

`passwd [ name ]`

DESCRIPTION

The command `passwd` changes or installs a password associated with the login `name`.

Ordinary users may change only the password which corresponds to their login `name`.

The command `passwd` prompts ordinary users for their old password, if any. It then prompts for the new password twice. If password aging is in effect, then the first time the new password is entered, `passwd` checks to see if the old password has "aged" sufficiently. If "aging" is insufficient the new password is rejected and `passwd` terminates.

If "aging" is sufficient, a check is made to insure that the new password meets construction requirements. When the new password is entered a second time, the two copies of the new password are compared. If the two copies are not identical the cycle of prompting for the new password is repeated for at most two more times.

The super-user may change any password; hence, `passwd` does not prompt the super-user for the old password. The super-user is not forced to comply with password aging and password construction requirements. The super-user can create a null password by entering a carriage return in response to the prompt for a new password.

FILES

`/etc/passwd`

USAGE

End-user.

LEVEL

Level 1.

NAME

shl — shell layer manager

SYNOPSIS

`shl`

DESCRIPTION

The command `shl` allows a user to interact with more than one shell from a single terminal. The user controls these shells, known as *layers*, using the commands described below.

The *current layer* is the layer which can receive input from the keyboard. Other layers attempting to read from the keyboard are blocked. Output from multiple layers is multiplexed onto the terminal. To have the output of a layer blocked when it is not current, the `stty` option `loblk` may be set within the layer.

The `stty` character `swtch` (set to control-Z if NUL) is used to switch control to `shl` from a layer. The command `shl` has its own prompt, `>>>`, to help distinguish it from a layer.

A *layer* is a shell which has been bound to a virtual tty device (`/dev/sxt/*`). The virtual device can be manipulated like a real tty device using `stty` and `ioctl()`. [See STTY(AU_CMD) and IOCTL(BA_OS) respectively.] Each layer has its own process group ID.

Definitions

A *name* is a sequence of characters delimited by a blank, tab or newline. Only the first eight characters are significant. The *names* **(1)** through **(7)** cannot be used when creating a layer. They are used by `shl` when no name is supplied. They may be abbreviated to just the digit.

Commands

The following commands may be issued from the `shl` prompt level. Any unique prefix is accepted.

`create [` *name* `]`

Create a layer called *name* and make it the current layer. If no argument is given, a layer will be created with a name of the form `(#)` where `#` is the last digit of the virtual device bound to the layer. The shell prompt variable `PS1` is set to the name of the layer followed by a space. A maximum of seven layers can be created.

`block` *name* `[` *name* `...]`

For each *name*, block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the `stty` option `loblk` within the layer.

`delete` *name* `[` *name* `...]`

For each *name*, delete the corresponding layer. All processes in

the process group of the layer are sent the SIGHUP signal.

`help` (or `?`)
Print the syntax of the `shl` commands.

`layers [ -l ] [` *name* `... ]`
For each *name*, list the layer name and its process group. The `-l` option produces a long listing. If no arguments are given, information is presented for all existing layers.

`resume [` *name* `]`
Make the layer referenced by *name* the current layer. If no argument is given, the last existing current layer will be resumed.

`toggle`
Resume the layer that was current before the last current layer.

`unblock` *name* `[` *name* `... ]`
For each *name*, do not block the output of the corresponding layer when it is not the current layer. This is equivalent to setting the `stty` option `-loblk` within the layer.

`quit`
Exit `shl`. All layers are sent the SIGHUP signal.

name
Make the layer referenced by *name* the current layer.

FILES

`/dev/sxt/*` Virtual tty devices

USAGE

General.

SEE ALSO

SH(BU_CMD), STTY(AU_CMD), IOCTL(BA_OS), SIGNAL(BA_OS).

LEVEL

Level 1.

New in System V Release 2.

NAME

stty — set the options for a terminal

SYNOPSIS

`stty [ -a ] [ -g ] [ options ]`

DESCRIPTION

The command `stty` sets certain terminal I/O options for the device that is its standard input; without arguments, it reports the settings of certain options; with the `-a` option, it reports all of the option settings; with the `-g` option, it reports current settings in a form that can be used as an argument to another `stty` command. Detailed information about the modes listed in the first five groups below may be found in IOCTL(BA_OS). Options in the last group are implemented using options in the previous groups. Note that many combinations of options make no sense, but no sanity checking is performed. The options are selected from the following:

Control Modes

`parenb` (`-parenb`)
enable (disable) parity generation and detection.

`parodd` (`-parodd`)
select odd (even) parity.

`cs5 cs6 cs7 cs8`
select character size.

`0`
hang up phone line immediately.

`number`
Set terminal baud rate to the number given, if possible. (All speeds are not supported by all hardware interfaces.)

`hupcl` (`-hupcl`)
hang up (do not hang up) modem connection on last close.

`hup` (`-hup`)
same as `hupcl` (`-hupcl`).

`cstopb` (`-cstopb`)
use two (one) stop bits per character.

`cread` (`-cread`)
enable (disable) the receiver.

`clocal` (`-clocal`)
assume a line without (with) modem control.

`loblk` (`-loblk`)
block (do not block) output from a non-current layer.

Input Modes

`ignbrk (-ignbrk)`
ignore (do not ignore) break on input.

`brkint (-brkint)`
signal (do not signal) INTR on break.

`ignpar (-ignpar)`
ignore (do not ignore) parity errors.

`parmrk (-parmrk)`
mark (do not mark) parity errors.

`inpck (-inpck)`
enable (disable) input parity checking.

`istrip (-istrip)`
strip (do not strip) input characters to seven bits.

`inlcr (-inlcr)`
map (do not map) NL to CR on input.

`igncr (-igncr)`
ignore (do not ignore) CR on input.

`icrnl (-icrnl)`
map (do not map) CR to NL on input.

`iuclc (-iuclc)`
map (do not map) upper-case alphabetics to lower case on input.

`ixon (-ixon)`
enable (disable) START/STOP output control. Output is stopped by sending an ASCII DC3 and started by sending an ASCII DC1.

`ixany (-ixany)`
allow any character (only DC1) to restart output.

`ixoff (-ixoff)`
request that the system send (not send) START/STOP characters when the input queue is nearly empty/full.

Output Modes

`opost (-opost)`
post-process output (do not post-process output; ignore all other output modes).

`olcuc (-olcuc)`
map (do not map) lower-case alphabetics to upper case on output.

`onlcr (-onlcr)`
map (do not map) NL to CR-NL on output.

`ocrnl (-ocrnl)`
map (do not map) CR to NL on output.

`onocr (-onocr)`
do not (do) output CRs at column zero.

`onlret (-onlret)`
on the terminal NL performs (does not perform) the CR function.

`ofill (-ofill)`
use fill characters (use timing) for delays.

`ofdel (-ofdel)`
fill characters are DELs (NULs).

`cr0 cr1 cr2 cr3`
select style of delay for carriage returns.

`nl0 nl1`
select style of delay for line-feeds.

`tab0 tab1 tab2 tab3`
select style of delay for horizontal tabs.

`bs0 bs1`
select style of delay for backspaces.

`ff0 ff1`
select style of delay for form-feeds.

`vt0 vt1`
select style of delay for vertical tabs.

Local Modes

`isig (-isig)`
enable (disable) the checking of characters against the special control characters INTR, QUIT, and SWTCH.

`icanon (-icanon)`
enable (disable) canonical input (ERASE and KILL processing).

`xcase (-xcase)`
canonical (unprocessed) upper/lower-case presentation.

`echo (-echo)`
echo back (do not echo back) every character typed.

`echoe (-echoe)`
echo (do not echo) ERASE character as a backspace-space-backspace string. Note: this mode will erase the ERASEed character on many CRT terminals; however, it does *not* keep track of column position and, as a result, may be confusing on escaped characters, tabs, and backspaces.

`echok (-echok)`
echo (do not echo) NL after KILL character.

`lfkc (-lfkc)`
the same as `echok (-echok)`; obsolete.

`echonl (-echonl)`
echo (do not echo) NL.

`noflsh (-noflsh)`
disable (enable) flush after INTR, QUIT, or SWTCH.

Control Assignments

`control-character c`
set `control-character` to `c`, where `control-character` is `erase`, `kill`, `intr`, `quit`, `swtch`, `eof`, `eol`, `min`, or `time` (`min` and `time` are used with `-icanon`). If `c` is preceded by a caret (^), then the value used is the corresponding CTRL character (e.g., "^d" is a **CTRL-d**); "^?" is interpreted as DEL and "^-" is interpreted as undefined.

`line i`
set line discipline to `i` (0 < `i` < 127).

Combination Modes

`evenp` or `parity`
enable `parenb` and `cs7`.

`oddp`
enable `parenb`, `cs7`, and `parodd`.

`-parity`, `-evenp`, or `-oddp`
disable `parenb`, and set `cs8`.

`raw (-raw` or `cooked)`
enable (disable) raw input and output (no ERASE, KILL, INTR, QUIT, SWTCH, EOT, or output post processing).

`nl (-nl)`
unset (set) `icrnl`, `onlcr`. In addition `-nl` unsets `inlcr`, `igncr`, `ocrnl`, and `onlret`.

`lcase (-lcase)`
set (unset) `xcase`, `iuclc`, and `olcuc`.

`LCASE (-LCASE)`
same as `lcase (-lcase)`.

`tabs (-tabs` or `tab8)`
preserve (expand to spaces) tabs when printing.

`ek`
reset ERASE and KILL characters back to normal # and @.

`sane`
resets all modes to some reasonable values.

USAGE

End-user.

SEE ALSO

IOCTL(BA_OS).

LEVEL

Level 1.

NAME

su — become super-user or another user

SYNOPSIS

```
su [ - ] [ name [ arg ... ] ]
```

DESCRIPTION

The command `su` allows one to become another user without logging off. The default user `name` is `root` (i.e., super-user).

To use `su`, the appropriate password must be supplied (unless one is already `root`). If the password is correct, `su` will execute a new environment with the real and effective user ID set to that of the specified user. The new command interpreter will be the optional program named in the specified user's password file entry, or the default if none is specified. Normal user ID privileges can be restored by entering EOT (control-D).

Any additional arguments given on the command line are passed to the command interpreter.

The following statements are true only if the command interpreter named in the specified user's password file entry is `sh` [see SH(BU_CMD)]. If the first argument to `su` is a `-`, the environment will be changed to what would be expected if the user actually logged in as the specified user. Otherwise, the environment is passed along with the possible exception of `PATH`.

All attempts to become another user using `su` are logged.

FILES

`/etc/passwd`	system's password file
`/etc/profile`	system's profile
`$HOME/.profile`	`user's profile`

USAGE

General.

SEE ALSO

SH(BU_CMD).

LEVEL

Level 1.

NAME

tabs — set tabs on a terminal

SYNOPSIS

`tabs [ tabspec ] [ +mn ] [ -Ttype ]`

DESCRIPTION

The command `tabs` sets the tab stops on the user's terminal according to the tab specification `tabspec`, after clearing any previous settings.

Three types of tab specification are accepted for `tabspec`: "canned," repetitive, arbitrary. If no `tabspec` is given, the default value is `-8`, i.e., "standard" tabs. The lowest column number is 1. Note that for `tabs`, column 1 always refers to the leftmost column on a terminal, even one whose column markers begin at 0.

`-code` Gives the name of one of a set of "canned" tabs. The legal codes and their meanings are as follows:

`-a` 1,10,16,36,72
Assembler, IBM System/370, first format

`-a2` 1,10,16,40,72
Assembler, IBM System/370, second format

`-c` 1,8,12,16,20,55
COBOL, normal format

`-c2` 1,6,10,14,49
COBOL compact format (columns 1-6 omitted).

`-c3` 1,6,10,14,18,22,26,30,34,38,42,46,50,54,58,62,67
COBOL compact format (columns 1-6 omitted), with more tabs than `-c2`. This is the recommended format for COBOL.

`-f` 1,7,11,15,19,23
FORTRAN

`-p` 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/I

`-s` 1,10,55
SNOBOL

`-u` 1,12,20,44
UNIVAC 1100 Assembler

In addition to these "canned" formats, three other types exist:

`-n` A repetitive specification requests tabs at columns 1+n, 1+2*n, etc. Of particular importance is the value `-8`: this represents the "standard" tab setting, and is the most likely tab setting to be found at a terminal. Another special case is the value `-0`, implying no tabs at all.

`n1,n2,...` The arbitrary format permits the user to type any chosen set of numbers, separated by commas, in ascending order. Up to 40 numbers are allowed. If any number (except the first one) is preceded by a plus sign, it is taken as an increment to be added to the previous value. Thus, the tab lists

1,10,20,30 and 1,10,+10,+10 are considered identical.

Any of the following may be used also; if a given flag occurs more than once, the last value given takes effect:

`-Ttype` The command `tabs` usually needs to know the type of terminal in order to set tabs and always needs to know the type to set margins. The argument `type` is a terminal name. If no `-T` flag is supplied, `tabs` searches for the environmental variable `TERM`. If no `type` can be found, `tabs` tries a sequence that will work for many terminals.

`+mn` The margin argument may be used for some terminals. It causes all tabs to be moved over `n` columns by making column `n+1` the left margin. If `+m` is given without a value of `n`, the value assumed is 10. For a TermiNet, the first value in the tab list should be 1, or the margin will move even further to the right. The normal (leftmost) margin on most terminals is obtained by `+m0`. The margin for most terminals is reset only when the `+m` flag is given explicitly.

Tab and margin setting is performed via the standard output.

USAGE

End-user.

LEVEL

Level 1.

NAME

tar — file archiver

SYNOPSIS

`tar [ option ] [ file ... ]`

DESCRIPTION

The command `tar` creates archives of files; it is often used to save files on (and restore from) magnetic tape. Its actions are controlled by the `option` argument. The `option` is a string of characters containing at most one function letter and possibly one or more modifiers. Other arguments to the command are `files` (or directory names) specifying which files are to be archived or restored. In all cases, appearance of a directory name refers to the files and (recursively) subdirectories of that directory.

The function portion of the option is specified by one of the following letters:

`r` The named `files` are written on the end of the archive.

`x` The named `files` are extracted from the archive. If a named file matches a directory whose contents had been written onto the archive, this directory is (recursively) extracted. If a named file in the archive does not exist on the system, the file is created with the same mode as the one in the archive, except that the set-user-ID and set-group-ID modes are not set unless the user is super-user. If the files exist, their modes are not changed except as described above. The owner, group, and modification time are restored (if possible). If no `files` argument is given, the entire content of the archive is extracted. Note that if several files with the same name are in the archive, the last one overwrites all earlier ones.

`t` The names of all the files in the archive are listed.

`u` The named `files` are added to the archive if they are not already there, or have been modified since last written into the archive. This option implies option `r`.

`c` Create a new archive; writing begins at the beginning of the archive, instead of after the last file. This option implies the `r` option.

The following characters may be used in addition to the letter that selects the desired function:

`v` Normally, `tar` does its work silently. The `v` (verbose) modifier causes it to type the name of each file it treats, preceded by the option letter. With the `t` option, `v` gives more information about the archive entries than just the name.

`w` Causes `tar` to print the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If a word beginning with `y` is given, the action is performed. Any other input means "no". This modifier is invalid with the `t` option.

`f` Causes `tar` to use the next argument as the name of the archive instead of the default, which is usually a tape drive. If the name of the file is `-`, `tar` writes to the standard output or reads from the standard input, whichever is appropriate. Thus, `tar` can be used as the head or tail of a pipeline. The command `tar` can also be used to move directory hierarchies with the command:

```
(cd fromdir; tar cf - .) | (cd todir; tar xf -)
```

`b` Causes tar to use the next argument as the blocking factor for tape records. The default is 1, the maximum is 20. This option should only be used with (raw) magnetic tape archives (see `f` above). The block size is determined automatically when reading tapes (options `x` and `t`).

`l` Tells `tar` to report if it cannot resolve all of the links to the files being archived. If `l` is not specified, no error messages are printed. This modifier is valid only with the options `c`, `r`, and `u`.

`m` Tells `tar` not to restore the modification times. The modification time of the file will be the time of extraction. This modifier is invalid with the `t` option.

`o` Causes extracted files to take on the user and group identifier of the user running the program rather than those on the archive. This modifier is valid only with the `x` option.

ERRORS

The command `tar` reports bad option characters and read/write errors. It also reports an error if enough memory is not available to hold the link tables.

USAGE

General.

LEVEL

Level 1.

NAME

tty – get the name of the terminal

SYNOPSIS

`tty [ -s ]`

DESCRIPTION

The command `tty` prints the path name of the user's terminal. The `-s` option inhibits printing of the terminal path name, allowing one to test just the exit code.

ERRORS

Exit codes:

2 if invalid options were specified,
0 if standard input is a terminal,
1 otherwise.

An error is reported if the standard input is not a terminal and `-s` is not specified.

USAGE

General.

LEVEL

Level 1.

NAME

uucp, uulog, uuname — system-to-system copy

SYNOPSIS

```
uucp [ options ] source-files destination-file

uulog [ -s system ]

uuname [ -l ]
```

DESCRIPTION

uucp

The command `uucp` copies files named by the `source-file` arguments to the `destination-file` argument. (Note that some `uucp` options are new to Release 2; see the options paragraph below for details.) A file name may be a path name on your machine, or may have the form:

```
system-name!path-name
```

where `system-name` is taken from a list of system names that `uucp` knows about. The destination `system-name` may also be a list of names such as

```
system-name!system-name!...!system-
name!path-name
```

in which case, an attempt is made to send the file via the specified route to the destination. Care should be taken to ensure that intermediate nodes in the route are willing to forward information.

The shell metacharacters ?, *, and [...] appearing in `path-name` will be expanded on the appropriate system.

Path-names may be one of:

(1) a full path-name.

(2) a path-name preceded by ~`name` where `name` is a login name on the specified system and is replaced by that user's login directory. Note that if an invalid login is specified, the default will be to the public directory (PUBDIR).

(3) a path-name specified as ~/`dest`, where the destination `dest` is appended to PUBDIR.

NOTE: This destination will be treated as a file name unless more than one file is being transferred by this request or the destination is already a directory. To ensure that it is a directory, follow the destination with a '/'. For example, ~/dan/ as the destination will make the directory PUBDIR/dan if it does not exist and put the requested file(s) in that directory.

(4) anything else is prefixed by the current directory.

If the result is an erroneous path-name for the remote system, the copy will fail. If the `destination-file` is a directory, the last part of the `source-file` name is used.

The command `uucp` gives universal read and write permissions and preserves execute permissions across the transmission.

The following options are interpreted by `uucp`:

`-c` Do not copy local file to the spool directory for transfer to the remote machine (default).

`-C` Force the copy of local files to the spool directory for transfer.

`-d` Make all necessary directories for the file copy (default).

`-f` Do not make intermediate directories for the file copy.

`-j` Output the job identification ASCII string on the standard output. This job identification can be used by `uustat` to obtain the status or terminate a job. (This option is new to System V Release 2.)

`-m` Send mail to the requester when the copy is completed.

`-nuser` Notify `user` on the remote system that a file was sent.

`-r` Do not start the file transfer; just queue the job. (This option is new to System V Release 2.)

uulog

The command `uulog` queries a log file of `uucp` or `uuxqt` transactions.

If the `-s` option is specified, then `uulog` prints information about file transfer work involving system `system`.

uuname

The command `uuname` lists the `uucp` names of known systems. The `-l` option returns the local system name.

USAGE

General.

The domain of remotely accessible files can (and for obvious security reasons, usually should) be severely restricted.

SEE ALSO

MAIL(BU_CMD), UUSTAT(AU_CMD), UUX(AU_CMD).

LEVEL

Level 1.

NAME

uustat — uucp status inquiry and job control

SYNOPSIS

`uustat [ options ]`

DESCRIPTION

The command `uustat` will display the status of, or cancel, previously specified `uucp` commands, or provide general status on `uucp` connections to other systems.

Not all combinations of options are valid. Only one of the following options can be specified with `uustat`:

`-q` List the jobs queued for each machine.

`-k` *jobid* Kill the `uucp` request whose job identification is *jobid*. The killed `uucp` request must belong to the person issuing the `uustat` command unless that user is the superuser.

`-r` *jobid* Rejuvenate *jobid*. The files associated with *jobid* are touched so that their modification time is set to the current time. This prevents the cleanup daemon from deleting the job until the jobs modification time reaches the limit imposed by the daemon.

The options below may not be used with the ones listed above; however, these options may be used singly or together:

`-s` *sys* Report the status of all `uucp` requests for remote system *sys*.

`-u` *user* Report the status of all `uucp` requests issued by *user*.

When no options are given, `uustat` outputs the status of all `uucp` requests issued by the current user.

USAGE

General.

SEE ALSO

UUCP(AU_CMD).

LEVEL

Level 1.

NAME

uuto, uupick — public system-to-system file copy

SYNOPSIS

```
uuto [ -p ] [ -m ] source-files destination

uupick [ -s system ]
```

DESCRIPTION

uuto

The command `uuto` sends `source-files` to `destination`. The command `uuto` uses the UUCP(AU_CMD) facility to send files, while it allows the local system to control the file access. A source-file name is a path name on the user's machine. Destination has the form: "`system!user`" where `system` is taken from a list of system names that `uucp` knows about [see `uuname` in UUCP(AU_CMD).] The argument `user` is the login name of someone on the specified system.

Two `options` are available:

`-p` Copy the source file into the spool directory before transmission.
`-m` Send mail to the sender when the copy is complete.

The files (or subtrees if directories are specified) are sent to a public directory (PUBDIR) on `system`. Specifically, the files are sent to the directory

PUBDIR`/receive/user/fsystem`,
where `user` is the recipient, and `fsystem` is the sending system.

The recipient is notified by `mail` of the arrival of files.

uupick

The command `uupick` may be used by a user to accept or reject the files transmitted to the user. Specifically, `uupick` searches PUBDIR on the user's system for files sent to the user. For each entry (file or directory) found, one of the following messages is printed on the standard output:

```
from system: dir dirname ?

from system: file file-name ?
```

The command `uupick` then reads a line from the standard input to determine the disposition of the file. The user's possible responses are:

<newline> Go on to next entry.

`d` Delete the entry.

`m` [*dir*] Move the entry to named directory `dir`. If `dir` is not specified as a complete path name a destination relative to the current directory is assumed. If no destination is given, the default is the current directory.

`a [` *dir* `]`	Same as `m` except moving all the files sent from `system`.
`p`	Print the content of the file to standard output.
`q`	Stop and exit.
EOT	(control-D.) Same as `q`.
`!`*command*	Escape to the command interpreter to execute *command*.
`*`	Print a usage summary for `uuto`.

The command `uupick` invoked with the `-s system` option will only search for files (and list any found) sent from `system`.

USAGE

General.

SEE ALSO

MAIL(BU_CMD), UUCP(AU_CMD), UUSTAT(AU_CMD), UUX(AU_CMD),

LEVEL

Level 1.

NAME

uux — remote command execution

SYNOPSIS

```
uux [ options ] command-string
```

DESCRIPTION

The command `uux` will gather zero or more files from various systems, execute a command on a specified system, and then send the standard output of the command to a file on a specified system.

The `command-string` is made up of one or more arguments that are similar to normal command arguments, except that the command and any file names may be prefixed by `system-name!`. A null `system-name` is interpreted as the local system.

The following statements are relevant if SH(BU_CMD) is the command interpreter.

The metacharacter * will not give the desired result.

The redirection tokens >> and << are not implemented.

A file name may be specified as for `uucp`: it may be a full path name, a path name preceded by `~name` (which is replaced by the corresponding login directory), a path name specified as `~/dest` (`dest` is prefixed by PUBDIR), or a simple file name (which is prefixed by the current directory). See UUCP(AU_CMD) for the details.

As an example, the command

```
uux "!diff usg!/usr/dan/file1
pwba!/a4/dan/file2 >!~/dan/file.diff"
```

will get the `file1` and `file2` files from the "usg" and "pwba" machines, execute `diff`, and put the results in `file.diff` in the local PUBDIR/dan directory. (PUBDIR is the `uucp` public directory on the local system.)

The execution of commands on remote systems takes place in an execution directory known to the `uucp` system. All files required for the execution will be put into this directory unless they already reside on that machine. Therefore, the non-local file names (without path or machine reference) must be unique within the `uux` request. The following command will **not** work:

```
uux "a!diff b!/usr/dan/xyz c!/usr/dan/xyz
>!xyz.diff"
```

because the file `xyz` will be copied from the `b` system as well as the `c` system, causing a name conflict. The command

```
uux "a!diff a!/usr/dan/xyz c!/usr/dan/xyz
>!xyz.diff"
```

will work (provided `diff` is a permitted command), because the local file `xyz` (which is not copied) does not conflict with the copied file `xyz` from the `c` system.

Any characters special to the command interpreter should be quoted either by quoting the entire `command-string` or quoting the special characters as individual arguments.

The command `uux` will attempt to get all files to the execution system. For files that are output files, the file name must be escaped using parentheses. For example, the command

```
uux a!cut -f1 b!/usr/file \(c!/usr/file\)
```

gets /usr/`file` from system "b", sends it to system "a", performs a `cut` command on that file, and sends the result of the `cut` command to system "c".

The command `uux` will notify the user (by mail) if the requested command on the remote system was disallowed. This notification can be turned off by the `-n` option. The response comes by mail from the remote machine.

The following options are interpreted by `uux`:

- The standard input to `uux` is made the standard input to the `command-string`.

`-j` Output the job identification ASCII string on the standard output. This job identification can be used by `uustat` to obtain the status or terminate a job. (This option is new to Release 2.)

`-n` Do not notify the user if the command fails.

USAGE

General.

Note that, for security reasons, many installations will limit the list of commands executable on behalf of an incoming request from `uux`. Many sites will permit little more than the receipt of mail via `uux`.

Only the first command of a pipeline [see SH(BU_CMD)] may have a `system-name!`. All other commands are executed on the system of the first command.

SEE ALSO

UUCP(AU_CMD), UUSTAT(AU_CMD).

LEVEL

Level 1.

NAME

vi — screen-oriented (visual) display editor

SYNOPSIS

```
vi [ -r file ] [ -l ] [ -wn ] [ -R ] [ +command ]
file ...
```

DESCRIPTION

`vi` (visual) is a display-oriented text editor. It is based on the underlying line editor EX(AU_CMD): it is possible to switch back and forth between the two, and to execute `ex` commands from within `vi`.

When using `vi`, the terminal screen acts as window into the file being edited. Changes made to the file are reflected in the screen display; the position of the cursor on the screen indicates the position within the file.

The environmental variable `TERM` must give the terminal type; the terminal must be defined in the *terminfo* database. As for `ex`, editor initialization scripts can be placed in the environmental variable `EXINIT`, or the file `.exrc` in the current or home directory.

Options

The following options are interpreted by `vi`:

`-r`*file*	Recover *file* after an editor or system crash. If *file* is not specified a list of all saved files will be printed.
`-l`	set **LISP** mode (see **Edit Options** below).
`-w`*n*	Set the default window size to *n*.
`-R`	Read only mode; the **readonly** flag is set, preventing accidental overwriting of the file.
`+`*command*	The specified `ex` command is interpreted before editing begins.

VI COMMANDS

General Remarks

See EX(AU_CMD) for the complete description of `ex`. Only the `visual` mode of the editor is described here.

At the beginning, `vi` is in the `command` mode; the `input` mode is entered by several commands used to insert or change text. In input mode, ESC (escape) is used to leave input mode; in command mode, it is used to cancel a partial command; the terminal bell is sounded if the editor is not in input mode and there is no partially entered command.

The last (bottom) line of the screen is used to echo the input for search commands (/ and ?), for `ex` commands (:), and system commands (!). It is also used to report errors or print other messages.

An interrupt (BREAK or DEL) typed during text input, or during the input of a command on the bottom line, terminates the input (or cancels the command) and returns the editor to command mode. During command mode an interrupt causes the bell to be sounded; in general the bell indicates an error (such as unrecognized key).

Lines displayed on the screen containing only a '~' indicate that the last line above them is the last line of the file (the '~' lines are past the end of the file). On a terminal with limited local intelligence, there may be lines on the screen marked with an '@': these indicate space on the screen not corresponding to lines in the file. (These lines may be removed by entering a 'control-R', forcing the editor to retype the screen without these holes.)

Command Summary

Most commands accept a preceding number as an argument, either to give a size or position (for display or movement commands), or as a repeat count (for commands that change text). For simplicity, this optional argument will be referred to as `count` when its effect is described.

The following operators can be followed by a movement command, in order to specify an extent of text to be affected: `c`, `d`, `y`, `<`, `>`, `!`, and `=`. The region specified is from the current cursor position to just before the cursor position indicated by the move. If the command operates on lines only, then all the lines which fall partly or wholly within this region are affected. Otherwise the exact marked region is affected.

In the following, control characters are indicated in the form '^X', which stands for 'control-X'. The intended ASCII character name is also given.

Unless otherwise specified, the commands are interpreted in command mode and have no special effect in input mode.

^B (STX) Scrolls backward to display the window above the current one. A count specifies the number of windows to go back. Two lines of overlap are kept if possible.

^D (EOT) Scrolls forward a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future **^D** and **^U** commands.

In input mode, backs `shiftwidth` spaces over the indentation provided by `autoindent` or **^T**.

^E (ENQ) Scrolls forward one line, leaving the cursor where it is if possible.

^F (ACK) Scrolls forward to display the window below the current one. A count specifies the number of windows to go

forward. Two lines of overlap are kept if possible.

^G (BEL) Prints the current file name and other information, including the number of lines and the current position. (Equivalent to the **ex** command **f**.)

^H (BS) Moves one space to the left (stops at the left margin). A count specifies the number of spaces to back up. (Same as **h**.)

In input mode, backs over the last input character without erasing it.

^J (LF) Moves the cursor down one line in the same column. A count specifies the number of lines to move down. (Same as ^N and **j**.)

^L (FF) Clears and redraws the screen. (Used when the screen is scrambled for any reason.)

^M (CR) Moves to the first non-white character in the next line. A count specifies the number of lines to go forward.

^N (SO) Same as ^J and **j**.

^P (DLE) Moves the cursor up one line in the same column. A count specifies the number of lines to move up. (Same as **k**.)

^R (DC2) Redraws the current screen, eliminating the false lines marked with '@' (which do not correspond to actual lines in the file).

^T (DC4) In input mode, if at the beginning of the line or preceded only by white space, inserts **shiftwidth** white space. This inserted space can only be backed over using ^D.

^U (NAK) Scrolls up a half-window of text. A count gives the number of (logical) lines to scroll, and is remembered for future ^D and ^U commands.

^V (SYN) In input mode, quotes the next character to make it possible to insert special characters (including ESC) into the file.

^W (ETB) In input mode, backs up one word; the deleted characters remain on the display.

^Y (EM) Scrolls backward one line, leaving the cursor where it is if possible.

^[(ESC) Cancels a partially formed command; sounds the bell if there is none.

In input mode, terminates input mode.

When entering a command on the bottom line of the screen (**ex** command line or search pattern with \ or ?),

terminates input and executes command.

SPACE Moves one space to the right (stops at the end of the line). A count specifies the number of spaces to go forward. (Same as **l**.)

! An operator which passes specified lines from the buffer as standard input to the specified system command, and replaces those lines with the standard output from the command. The **!** is followed by a movement command specifying the lines to be passed (lines from the current position to the end of the movement) and then the command (terminated as usual by a return). A count preceding the **!** is passed on to the movement command after **!**.

Doubling **!** and preceding it by a count causes that many lines, starting with the current line, to be passed.

" Precedes a named buffer specification. There are named buffers **1-9** in which the editor places deleted text. The named buffers **a-z** are available to the user for saving deleted or yanked text.

$** Moves to the end of the current line. A count specifies the number of lines to go forward. (e.g., **2$ goes to the end of the next line.)

% Moves to the parenthesis or curly brace which matches the parenthesis or brace at the current cursor position.

& Same as the **ex** command **&** (repeats previous substitute command).

' When followed by a **'**, returns to the previous context, placing the cursor at the beginning of the line. (The previous context is set whenever a non-relative move is made.) When followed by a letter **a-z**, returns to the line marked with that letter (see the **m** command), at the first non-white character in the line.

When used with an operator such as **d** to specify an extent of text, the operation takes place over complete lines. (See also **`**.)

When followed by a **`**, returns to the previous context, placing the cursor at the character position marked. (The previous context is set whenever a non-relative move is made.) When followed by a letter **a-z**, returns to the line marked with that letter (see the **m** command), at the character position marked.

When used with an operator such as **d** to specify an extent of text, the operation takes place from the exact marked

place to the current position within the line. (See also '.)

[[Backs up to the previous section boundary. A section is defined by the value of the `sections` option. Lines which start with a formfeed (^L character) or { also stop [[.

If the option *lisp* is set, stops at each (at the beginning of a line.

]] Moves forward to a section boundary (see [[).

^ Moves to the first non-white position on the current line.

(Moves backward to the beginning of a sentence. A sentence ends at a . ! or ? which is followed by either the end of a line or by two spaces. Any number of closing)] " and ' characters may appear between the . ! or ? and the spaces or end of line. A count moves back that many sentences.

If the *lisp* option is set. moves to the beginning of a LISP s-expression. Sentences also begin at paragraph and section boundaries (see { and [[below).

) Moves forward to the beginning of a sentence. A count moves forward that many sentences. (See (.)

{ Moves back to the beginning of the preceding paragraph. A paragraph is defined by the value of the `paragraphs` option. A completely empty line, and a section boundary (see [[above), are also taken to begin paragraphs. A count specifies the number of paragraphs to move backward.

} Moves forward to the beginning of the next paragraph. A count specifies the number of paragraphs to move forward. (See {.)

| Requires a count; the cursor is placed in that column (if possible).

+ Moves to the first non-white character in the next line. A count specifies the number of lines to go forward. (Same as ^M.)

, Reverse of the last **f F t** or **T** command, looking the other way in the current line. A count is equivalent to repeating the search that many times.

- Moves to the first non-white character in the previous line. A count specifies how many lines to move back.

. Repeats the last command which changed the buffer. A count is passed on to the command being repeated.

/ Reads a string from the last line on the screen, interprets it as a regular expression, and scans forward for the next

occurrence of a matching string. The search begins when return is entered to terminate the pattern; it may be terminated with an interrupt (or DEL).

When used with an operator to specify an extent of text, the defined region is from the current cursor position to the beginning of the matched string. Whole lines may be specified by giving an offset from the matched line (using a closing / followed by a +n or -n).

Regular expressions are described in EX(AU_CMD).

0 Moves to the first character on the current line. (Is not interpreted as a command when preceded by a non-zero digit.)

: Begins an `ex` command. The :, as well as the entered command, is echoed on the bottom line; It is executed when the input is terminated by entering a return.

; Repeats the last single character find using **f F t** or **T**. A count is equivalent to repeating the search that many times.

< An operator which shifts lines left one `shiftwidth`. May be followed by a move to specify lines. A count is passed through to the move command.

When repeated (**<<**), shifts the current line (or count lines starting at the current one).

> An operator which shifts lines right one `shiftwidth`. (See **<**.)

= If the `lisp` option is set, then reindents the specified lines, as though they were typed in with *lisp* and *autoindent* set. May be preceded by a count to indicate how many lines to process, or followed by a move command for the same purpose.

? Scans backwards, the reverse of **/**. (See **/**.)

A Appends at the end of line. (Same as **$a**.)

B Backs up a word, where a word is any non-blank sequence, placing the cursor at the beginning of the word. A count gives the number of words to go back.

C Changes the rest of the text on the current line. (Same as **c$**.)

D Deletes the rest of the text on the current line. (same as **d$**.)

E Moves forward to the end of a word, where a word is any non-blank sequence. A count gives the number of words to

go forward.

F Must be followed by a single character; scans backwards in the current line for that character, moving the cursor to it if found. A count is equivalent to repeating the search that many times.

G Goes to the line number given as preceding argument, or the end of the file if no preceding count is given.

H Moves the cursor to the top line on the screen. If a count is given, then the cursor is moved to that line on the screen, counting from the top. The cursor is placed on the first non-white character on the line. If used as the target of an operator, full lines are affected.

I Inserts at the beginning of a line. (Same as **↑i**.)

J Joins the current line with the next one, supplying appropriate white space: one space between words, two spaces after a period, and no spaces at all if the first character of the next line is). A count causes that many lines to be joined rather than two.

L Moves the cursor to the first non-white character of the last line on the screen. A count moves to that line counting form the bottom. When used with an operator, whole lines are affected.

M Moves the cursor to the middle line on the screen, at the first non-white position on the line.

N Scans for the next match of the last pattern given to / or ?, but in the reverse direction; this is the reverse of **n**.

O Opens a new line above the current line and enters input mode.

P Puts the last deleted text back before/above the cursor. The text goes back as whole lines above the cursor if it was deleted as whole lines. Otherwise the text is inserted just before the cursor.

May be preceded by a named buffer specification ("*x*), to retrieve the contents of the buffer.

Q Quits from *vi* and enters *ex* command mode.

R Replaces characters on the screen with characters entered, until the input is terminated with ESC.

S Changes whole lines (same as **cc**). A count changes that many lines.

T Must be followed by a single character; scans backwards in the current line for that character, and if found, places the cursor just after that character. A count is equivalent to repeating the search that many times.

U Restores the current line to its state before the cursor was last moved to it.

W Moves forward to the beginning of a word in the current line, where a word is a sequence of non-blank characters. A count specifies the number of words to move forward.

X Deletes the character before the cursor. A count repeats the effect, but only characters on the current line are deleted.

Y Places (yanks) a copy of the current line into the unnamed buffer (same as **yy**). A count copies that many lines. May be preceded by a buffer name to put the copied line(s) in that buffer.

ZZ Exits the editor, writing out the buffer if it was changed since the last write. (Same as the **ex** command **x**.)

a Enters input mode, appending the entered text after the current cursor position; A count causes the inserted text to be replicated that many times, but only if the inserted text is all on one line.

b Backs up to the beginning of a word in the current line. A word is a sequence of alphanumerics, or a sequence of special characters. A count repeats the effect.

c Deletes the specified region of text, and enters input mode to replace it with the entered text. If more than part of a single line is affected, the deleted text is saved in the numeric buffers. If only part of the current line is affected, then the last character to be deleted is marked with a $. A count is passed through to the move command.

d Deletes the specified region of text. If more than part of a line is affected, the text is saved in the numeric buffers. A count is passed through to the move command.

e Moves forward to the end of the next word, defined as for **b**. A count repeats the effect.

f Must be followed by a single character; scans the rest of the current line for that character, and moves the cursor to it if found. A count repeats the find that many times.

h Moves the cursor one character to the left. (Same as **^H**.) A count repeats the effect.

i Enters input mode, inserting the entered text before the cursor. (See **a**.)

j Moves the cursor one line down in the same column. (Same as ^**J** and ^**N**.)

k Moves the cursor one line up. (Same as ^**P**.)

l Moves the cursor one character to the right. (Same as SPACE.)

m Must be followed by a single lower case letter **x**; marks the current position of the cursor with that letter. The exact position is referred to by `**x**; the line is referred to by '**x**.

n Repeats the last **/** or **?** scanning commands.

o Opens a line below the current line and enters input mode; otherwise like **O**.

p Puts text after/below the cursor; otherwise like **P**.

r Must be followed by a single character; the character under the cursor is replaced by the specified one. (The new character may be a newline.) A count replaces each of the following count characters with the single character given.

s Deletes the single character under the cursor, and enters input mode; the entered text replaces the deleted character. A count specifies how many characters from the current line are changed. The last character to be changed is marked with a **$**, as for **c**.

t Must be followed by a single character; scans the rest of the line for that character. The cursor is moved to just before the character, if it is found. A count is equivalent to repeating the search that many times.

u Reverses the last change made to the current buffer. If repeated, will alternate between these two states, thus is its own inverse. When used after an insert which inserted text on more than one line, the lines are saved in the numeric named buffers.

w Moves forward to the beginning of the next word, where word is the same as in **b**. A count specifies how many words to go forward.

x Deletes the single character under the cursor. With a count deletes that many characters forward from the cursor position, but only on the current line.

y Must be followed by a movement command; the specifed text is copied (yanked) into the unnamed temporary buffer.

If preceded by a named buffer specification, "*x*, the text is placed in that buffer also.

z Redraws the screen with the current line placed as specified by the following character: return specifies the top of the screen, . the center of the screen, and - the bottom of the screen. A count may be given after the z and before the following character to specify the new screen size for the redraw. A count before the z gives the number of the line to place in the center of the screen instead of the default current line.

USAGE

General.

SEE ALSO

EX(AU_CMD).

LEVEL

Level 1.

NAME

wall — write to all users

SYNOPSIS

`/etc/wall`

DESCRIPTION

The command `wall` reads its standard input until an end-of-file. It then prints this message on the terminals of all users currently logged-in, preceded by:

Broadcast Message from *login-id*

The sender must be super-user to override any protections the users may have invoked [see MESG(AU_CMD)].

USAGE

Administrator.

The command `wall` is used to warn all users, typically prior to shutting down the system.

SEE ALSO

MESG(AU_CMD), WRITE(AU_CMD).

LEVEL

Level 1.

NAME

who — who is on the system

SYNOPSIS

```
who [ options ] [ file ]

who am i

who am I
```

DESCRIPTION

The command `who` can list the user's name, terminal line, login time, elapsed time since activity occurred on the line, and the process-ID of the command interpreter for each current system user. It examines the `/etc/utmp` file to obtain its information. If `file` is given, that file is examined instead.

The command `who` with the `am i` or `am I` option identifies the invoking user.

Except for the default `-s` option, the general format for output entries is:

```
name [state] line time activity pid [comment]
[exit]
```

With options, `who` can list logins, logoffs, reboots, and changes to the system clock, as well as other processes spawned by the `init` process. These options are:

`-u` This option lists only those users who are currently logged in. The `name` is the user's login name. The `line` is the name of the line as found in the directory `/dev`. The `time` is the time that the user logged in. The `activity` is the number of hours and minutes since activity last occurred on that particular line. A dot (`.`) indicates that the terminal has seen activity in the last minute and is therefore "current". If more than twenty-four hours have elapsed or the line has not been used since boot time, the entry is marked **old**. This field is useful when trying to determine whether a person is working at the terminal or not. The `pid` is the process-ID of the user's login process.

`-T` This option is the same as the `-u` option, except that the `state` of the terminal line is printed. The `state` describes whether someone else can write to that terminal. A `+` appears if the terminal is writable by anyone; a `-` appears if it is not. The super-user can write to all lines having a `+` or a `-` in the `state` field. If a bad line is encountered, a `?` is printed.

`-l` This option lists only those lines on which the system is waiting for someone to login. The `name` field is LOGIN in such cases. Other fields are the same as for user entries except that the `state` field does not exist.

`-H` This option will print column headings above the regular output. (This option is new in System V Release 2.)

`-q` This is a quick `who`, displaying only the names and the number of users currently logged on. When this option is used, all other options are ignored. (This option is new in System V Release 2.)

`-p` This option lists any other process which is currently active and has been previously spawned by `init`.

`-d` This option displays all processes that have expired and not been respawned by `init`. The `exit` field appears for dead processes and contains the termination and exit values of the dead process. This can be useful in determining why a process terminated.

`-b` This option indicates the time and date of the last reboot.

`-r` This option indicates the current `run-level` of the `init` process.

`-t` This option indicates the last change to the system clock.

`-a` This option processes `/etc/utmp` or the named `file` with all options turned on.

`-s` This option is the default and lists only the `name`, `line`, and `time` fields.

FILES

`/etc/utmp`
`/dev/tty*`

USAGE

General.

LEVEL

Level 1.

NAME

write — write to another user

SYNOPSIS

`write user [ terminal ]`

DESCRIPTION

The command `write` copies lines from the user's terminal to that of another user. When first called, it sends the message:

```
Message from sender-login-id (ttynn) [ date ]...
```

to the user addressed. When it has successfully completed the connection, it also sends two bells to the sender's terminal to indicate that what the sender is typing is being sent.

The recipient of the message should write back, by typing `write sender-login-id`, on receipt of the initial message. Whatever each user types (except for command escapes, see below) is printed on the other user's terminal, until an end-of-file or an interrupt is sent. At that point `write` writes "EOT" on the other terminal and exits. The recipient can also stop further messages from coming in by executing "mesg n".

To write to a user who is logged in more than once, the `terminal` argument may be used to indicate which terminal to send to (e.g., `tty00`); otherwise, the first writable instance of the user found in `/etc/utmp` is assumed and an informational message is written.

A user may deny or grant write permission by use of the `mesg` command. Certain commands disallow messages in order to prevent interference with their output. However, if the sender has super-user permissions, messages can be forced onto a write-inhibited terminal.

If the character ! is found at the beginning of a line, `write` calls the command interpreter to execute the rest of the line as a command.

ERRORS

The following errors are reported:

- the user addressed is not logged on.
- the user addressed denies write permission [see MESG(AU_CMD)].
- the user's terminal is set to `mesg n` and the recipient cannot respond.
- the recipient changes permission (`mesg n`) after `write` had begun.

FILES

`/etc/utmp`

USAGE

End-user.

SEE ALSO

MESG(AU_CMD), WHO(AU_CMD).

LEVEL

Level 1.

Part IV

Administered Systems Extension Definition

Chapter 7
Introduction

7.1 OVERVIEW

The Administered System Extension mostly comprises utilities used for system administration. Many of these are in fact restricted to the super-user.

The System V Base, the Kernel Extension, the Basic Utilities Extension, and the Advanced Utilities Extension, are prerequisites for the Administered System Extension.

7.2 DESCRIPTION

UTILITIES

acctcms	dodisk	mknod	sadp
acctcom	fsck	monacct	sar
acctcon1	fsdb	mount	setmnt
acctcon2	fuser	mvdir	shutacct
acctdisk	fwtmp	ncheck **	startup
acctmerg	grpck	nice	sync
accton	init	prctmp	sysdef
acctprc1	ipcrm	prdaily	timex
acctprc2	ipcs	prtacct	turnacct
acctwtmp	killall	pwck	umount
chargefee	labelit	runacct	unlink
ckpacct	lastlogin	sa1	volcopy
clri **	link	sa2	whodo
devnm	mkfs	sadc	wtmpfix
diskusg			

** Level 2: December 1, 1985.

7.3 SPECIAL PROCESSES

This section is intended to provide some background information about the system process spawner (**init**), and about how a user is logged in (**getty** and **login**). The description here is a general one; there may be minor differences between different implementations of System V.

7.3.1 INIT

The **init** process is invoked at system initialization, as one of the steps in the boot procedure; its primary role is to create processes according to entries in the file **/etc/inittab**.

One kind of entry in this file specifies how the *getty* program (see "Getty" section below) is to be executed on the individual terminal lines available for users to log in. Other entries control the initiation of autonomous processes required by any particular system.

The system administrator communicates with the **init** process by executing the `init` command [see INIT(AS_CMD)]. (It is important to keep in mind the distinction between the two; here **init** refers to the special system process, while `init` refers to the command that allows communication with the special system process.)

Init considers the system to be in a particular *run-level* at any given time. A *run-level* can be viewed as a software configuration of the system, where each configuration is defined by the collection of processes that are to be spawned. The specification of the *run-levels* (that is, the specification of the processes to be spawned by **init**) is defined in the `/etc/inittab` file. There are eight allowed *run-levels*, **0**–**6** and **s** (or **S**). The *run-level* may be changed when the administrator runs the `init` command.

When it is invoked at system initialization, the first thing the **init** process does is to look for the `/etc/inittab` file and see if there is an entry of the type `initdefault`. If there is, **init** uses the *run-level* specified in that entry as the initial *run-level* to enter. If this entry is not in `/etc/inittab`, then **init** requests that the user enter a *run-level* from the virtual system console, `/dev/console`.

The *run-level* **s** (**S**) corresponds to the *SINGLE USER* level. This is the only *run-level* that does not require the existence of a properly formatted `/etc/inittab` file. If `/etc/inittab` doesn't exist, then by default **init** enters the *SINGLE USER* level. (Note: Since other *run-levels* may also be configured for single-user, the *SINGLE USER* level need not be the only level in which only one user is allowed on the system.)

The levels **0** through **6** have no special meaning; they are defined by the entries in the `/etc/inittab` file.

Whenever a new *run-level* is entered, **init** scans the `/etc/inittab` file and processes all entries corresponding to that *run-level*. In addition, entry to and exit from the *SINGLE USER* level results in some special actions, as follows:

1. When the *SINGLE USER* level is entered in the boot sequence, **/etc/inittab** is scanned for any entries of the type `sysinit`. These entries are processed before any other actions in state "**s**".
2. The first time **init** leaves the *SINGLE USER* level, it scans **/etc/inittab** for special entries of the type `boot` and `bootwait`. Entries of this type, for which the specified *run-level* matches the new *run-level* to be entered, are

performed before any normal processing of **/etc/inittab** takes place.

In this way any special initialization of the operating system, such as mounting file systems, can take place before users are allowed onto the system.

In a normal operating environment (multi-user), the `/etc/inittab` file is usually set up so that **init** will create a process for each terminal on the system.

After it has spawned all of the processes specified by the `/etc/inittab` file, **init** waits for one of its child processes to die, a powerfail signal, or until a request is made via the `init` command. to change the system's *run-level*. When one of the above three conditions occurs, **init** re-examines the `/etc/inittab` file. New entries can be added to this file at any time; however, **init** still waits for one of the above three conditions to occur. (To get an immediate response the `init` command may be invoked with the option *q* in order to force **init** to re-examine the `/etc/inittab` file.

When **init** is requested to change *run-levels*, all processes defined in the current *run-level*, that are undefined in the target *run-level*, are terminated. This is done by first sending the signal **SIGTERM** (which serves as a warning for processes that catch it), and, after a brief delay, sending the signal **SIGKILL**.

If **init** finds that it is continuously respawning an entry from `/etc/inittab` (more frequently than some specified rate), it will assume that there is an error in the command string, generate an error message on the system console, and refuse to respawn this entry until either some time has elapsed or it receives a directive from the `init` command. This prevents **init** from eating up system resources when someone makes a typographical error in the `/etc/inittab` file or a program is removed that is referenced in `/etc/inittab`.

7.3.2 GETTY

Getty is a program that is invoked by the **init** process, to allow a user to login on a terminal line. It is thus the *getty* process that the user encounters when logging in to the system.

The actions of **getty** are controlled by entries in the file `/etc/gettydefs`. These entries specify what line speed should be used initially, what the login message should look like, what the initial tty settings are, and what speed to try next should the user indicate that the speed is inappropriate (by typing a **BREAK** character).

If a null character (or framing error) is received, it is assumed to be the result of the user pushing the "break" key. This will cause **getty** to attempt the next speed in a sequence defined in `/etc/gettydefs`.

Finally, the **login** command is called to allow the user to complete logging in.

7.3.3 LOGIN

The **login** process is invoked by the **getty** process, as described above, at the beginning of each terminal session. It is the means by which the user is identified to the system.

If the user has a password, **login** asks for it, and verifies its correctness. If system echoing has been enabled, it is turned off during the typing of the password. (However, echoing will continue to occur if local echo has been enabled).

If the login is not completed successfully within a certain period of time (e.g., one minute), the user may be disconnected.

After a successful login, the user-ID, the group-ID, the working directory, and the command interpreter, are initialized.

Chapter 8
Commands and Utilities

NAME

accton, acctwtmp, chargefee, ckpacct, dodisk, lastlogin, monacct, prdaily, prtacct, shutacct, startup, turnacct — miscellaneous accounting and support commands

SYNOPSIS

```
/usr/lib/acct/accton [ file ]

/usr/lib/acct/acctwtmp "reason"

/usr/lib/acct/chargefee login-name number

/usr/lib/acct/ckpacct [ blocks ]

/usr/lib/acct/dodisk [ files ]

/usr/lib/acct/lastlogin

/usr/lib/acct/monacct number

/usr/lib/acct/prdaily [ -l ] [ -c ] [ mmdd ]

/usr/lib/acct/prtacct file [ "heading" ]

/usr/lib/acct/shutacct [ "reason" ]

/usr/lib/acct/startup

/usr/lib/acct/turnacct on | off | switch
```

DESCRIPTION

The accounting software provides utilities to collect data on: process accounting, connect accounting, disk usage, command usage, summary command usage, and users' last login.

The `runnacct` [see RUNACCT(AS_CMD)] and `monacct` commands use the utilities listed here to produce daily and monthly summary files and reports that can be printed using `prdaily`; they use a number of intermediate files and support utilities that can also be used to tailor-make new accounting systems. Many of these utilities produce or manipulate "total accounting" (**tacct**) records which can be summarized by `acctmerg` [see ACCTMERG(AS_CMD)] and printed using `prtacct`.

The command `accton` without parameters turns process accounting off. If `file` is given, `accton` will turn accounting on. The argument `file` must be the name of an existing file (normally `/usr/adm/pacct`), to which the system appends process accounting records [see ACCT(KE_OS)].

The command `acctwtmp` writes a *utmp* structure in record to its standard output. The record contains the current time and a string of characters that describe the `reason`. A record type of ACCOUNTING is assigned. The argument `reason` must be a string of (11 or less) characters, numbers, **$**, or spaces. For example, the following are suggestions for use in startup and shutdown procedures, respectively:

```
acctwtmp "acctg on" >> /etc/wtmp
acctwtmp "acctg off" >> /etc/wtmp
```

The command `chargefee` is invoked to charge a `number` of units to `login-name`. An **ASCII tacct** record is written to `/usr/adm/fee`, to be merged with other accounting records by `acctmerg`.

The command `ckpacct` is typically initiated via `cron` It periodically checks the size of `/usr/adm/pacct`. If the size exceeds `blocks`, 1000 by default, `turnacct` will be invoked with argument `switch`. If the number of free disk blocks in the `/usr` file system falls below 500, `ckpacct` will automatically turn off the collection of process accounting records via the `off` argument to `turnacct`. The accounting will be activated again on the next invocation of `ckpacct` when at least this number of blocks is restored.

The command `dodisk` is typically invoked by `cron` to perform the disk accounting functions. By default, it will do disk accounting on the special files in `/etc/checklist`. If `files` are used, they should be the special file names of mountable filesystems; disk accounting will be done on these filesystems only.

The command `lastlogin` is invoked (typically by `runacct`) to update `/usr/adm/acct/sum/loginlog`, which shows the last date on which each person logged in.

The command `monacct` is typically invoked once each month. The argument `number` indicates which month or period it is. If `number` is not given, it defaults to the current month (01–12). This default is useful if `monacct` is to executed via `cron` on the first day of each month. The command `monacct` creates summary files in `/usr/adm/acct/fiscal`, restarts summary files in `/usr/adm/acct/sum`, and deletes the previous days' accounting reports (see `prdaily` below).

The command `prdaily` is invoked (typically by `runacct`) to format a report of the previous day's accounting data. The report resides in `/usr/adm/acct/sum/rprtmmdd` where `mmdd` is the month and day of the report. The current daily accounting reports may be printed by typing `prdaily`. Previous days' accounting reports can be printed by using the `mmdd` option and specifying the report date desired. The `-l` option prints a report of exceptional usage by login id for the specifed date. Previous daily reports are removed and therefore inaccessible after each invocation of `monacct`. The `-c` option prints a report of exceptional resource usage by command, and may be used on current day's accounting data only.

The command `prtacct` can be used to format and print any total accounting (**tacct**) file.

The command `shutacct` is typically invoked during a system shutdown (usually in `/etc/shutdown`) to turn process accounting off and append

a "reason" record to `/etc/wtmp`.

The command `startup` is typically called by the system initialization routine to turn on process accounting whenever the system is brought up.

The command `turnacct` is an interface to `accton` to turn process accounting **on** or **off**. The `switch` argument turns accounting off, moves the current `/usr/adm/pacct` to the next free name in `/usr/adm/pacctincr` (where `incr` is a number starting with 1 and incrementing by one for each additional `pacct` file), then turns accounting back on again.

FILES

`/etc/wtmp`	`login/logoff summary`
`/etc/passwd`	used for login name to user ID conversions
`/usr/lib/acct`	directory for accounting commands
`/usr/adm/fee`	accumulator for fees
`/usr/adm/pacct`	current file for process accounting
`/usr/adm/acct/sum`	summary directory

USAGE

Administrator.

SEE ALSO

ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), CRON(AU_CMD), DISKUSG(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

acctcms — command summary from per-process accounting records

SYNOPSIS

```
/usr/lib/acct/acctcms [options] files
```

DESCRIPTION

The command `acctcms` reads one or more `files`, normally in the form produced by ACCT(KE_OS). It adds all records for processes that executed identically-named commands, sorts them, and writes them to the standard output, normally using an internal summary format. The `options` are:

- `-a` Print output in ASCII rather than in the internal summary format. The output includes command name, number of times executed, total kcore-minutes, total CPU minutes, total real minutes, mean size (in K), mean CPU minutes per invocation, "hog factor", characters transferred, and blocks read and written, as in ACCTCOM(AS_CMD). Output is normally sorted by total kcore-minutes.
- `-c` Sort by total CPU time, rather than total kcore-minutes.
- `-j` Combine all commands invoked only once under "***other".
- `-n` Sort by number of command invocations.
- `-s` Any file names encountered hereafter are already in internal summary format.

The following options may be used only with the `-a` option.

- `-p` Output a prime-time-only command summary.
- `-o` Output a non-prime (offshift) time only command summary.

When `-p` and `-o` are used together, a combination prime and non-prime time report is produced. All the output summaries will be total usage except number of times executed, CPU minutes, and real minutes which will be split into prime and non-prime.

A typical sequence for performing daily command accounting and for maintaining a running total is:

```
acctcms file ... >today
cp total previoustotal
acctcms -s today previoustotal >total
acctcms -a -s today
```

USAGE

Administrator.

SEE ALSO

ACCT(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD), ACCTCOM(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

acctcom — search and print process accounting file(s)

SYNOPSIS

```
acctcom [ [ options ] [ file ] ] ...
```

DESCRIPTION

The command `acctcom` reads `file`, the standard input, or `/usr/adm/pacct`, in the form produced by ACCT(KE_OS) and writes selected records to the standard output. Each record represents the execution of one process and shows the **COMMAND NAME, USER, TTYNAME, START TIME, END TIME, REAL (SEC), CPU (SEC), MEAN SIZE(K),** and optionally, **F** (the `fork/exec` flag: **1** for `fork` without `exec`), **STAT** (the system exit status), **HOG FACTOR, KCORE MIN, CPU FACTOR, CHARS TRNSFD,** and **BLOCKS R/W** (total blocks read and written).

The command name is prepended with a **#** if it was executed with `super-user` privileges. If a process is not associated with a known terminal, a **?** is printed in the **TTYNAME** field.

If no `files` are specified, and if the standard input is associated with a terminal or `/dev/null`, `/usr/adm/pacct` is read; otherwise, the standard input is read.

If any `file` arguments are given, they are read in their respective order. Each file is normally read forward, i.e., in chronological order by process completion time. The `options` are:

`-a`	Show average statistics about the processes selected. The statistics will be printed after the output records.
`-b`	Read backwards, showing latest commands first. This `option` has no effect when the standard input is read.
`-f`	Print the `fork/exec` flag and system exit status columns in the output.
`-h`	Instead of mean memory size, show the fraction of total available CPU time consumed by the process during its execution (the "hog factor").
`-i`	Print columns containing total blocks read and written.
`-k`	Instead of memory size, show total kcore-minutes.
`-m`	Show mean core size (the default).
`-r`	Show CPU factor (user time/(system-time + user-time).
`-t`	Show separate system and user CPU times.
`-v`	Exclude column headings from the output.
`-l`*line*	Show only processes belonging to terminal `/dev/`*line*.
`-u`*user*	Show only processes belonging to *user* that may be specified by: a user ID, a login name that is then converted to a user ID, a # which designates only those processes executed with super-user privileges, or ? which designates only those processes associated with unknown user IDs.
`-g`*group*	Show only processes belonging to *group*. The *group* may be

designated by either the group ID or group name.

`-s`*time* Select processes existing at or after *time*, given in the format `hr[:min[:sec]]`.

`-e`*time* Select processes existing at or before *time*.

`-S`*time* Select processes starting at or after *time*.

`-E`*time* Select processes ending at or before *time*. Using the same `time` for both `-S` and `-E` shows the processes that existed at *time*.

`-n`*pattern* Show only commands matching *pattern* that may be a regular expression as in ED(BU_CMD) except that + means one or more occurrences.

`-q` Do not print any output records, just print the average statistics as with the `-a` option.

`-o`*ofile* Copy selected process records in the input data format to *ofile;* suppress standard output printing.

`-H`*factor* Show only processes that exceed *factor*, where factor is the "hog factor" as explained in option `-h` above.

`-O`*sec* Show only processes with CPU system time exceeding *sec* seconds.

`-C`*sec* Show only processes with total CPU time, system plus user, exceeding *sec* seconds.

`-I`*chars* Show only processes transferring more characters than the cut-off number given by *chars*.

FILES

/etc/passwd
/etc/group
/usr/adm/pacct

USAGE

Administrator.

SEE ALSO

ACCT(KE_OS), ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD).

LEVEL

Level 1.

NAME

acctcon1, acctcon2, prctmp — connect-time accounting

SYNOPSIS

```
/usr/lib/acct/acctcon1 [ options ]

/usr/lib/acct/acctcon2

/usr/lib/acct/prctmp
```

DESCRIPTION

The command `acctcon1` converts a sequence of login/logoff records read from its standard input to a sequence of session records, one per login session. Its input should normally be redirected from `/etc/wtmp`. The record format is ASCII, giving device, user ID, login name, prime connect time (seconds), non-prime connect time (seconds), session starting time (numeric), and starting date and time. The `options` are:

`-p` Print input only, showing line name, login name, and time (in both numeric and date/time formats).

`-t` `Acctcon1` maintains a list of lines on which users are logged in. When it reaches the end of its input, it emits a session record for each line that still appears to be active. It normally assumes that its input is a current file, so that it uses the current time as the ending time for each session still in progress. The `-t` flag causes it to use, instead, the last time found in its input, thus assuring reasonable and repeatable numbers for non-current files.

`-l` *file* *File* is created to contain a summary of line usage showing line name, number of minutes used, percentage of total elapsed time used, number of sessions charged, number of logins, and number of logoffs. This file helps track line usage, identify bad lines, and find software and hardware oddities. Various events during logoff each generate logoff records, so that the number of logoffs is often three to four times the number of sessions.

`-o` *file* *File* is filled with an overall record for the accounting period, giving starting time, ending time, and the count and type of various accounting records produced by `acctwtmp` [see ACCT(AS_CMD)].

The command `acctcon2` expects as input a sequence of login session records (as produced by `acctcon1`), and converts them into total accounting records.

The argument `Prctmp` can be used to print the session record file as produced by `acctcon1`.

EXAMPLES

These commands are typically used as shown below. The file `ctmp` can be used by `acctprc1` [see ACCTPRC(AS_CMD)]:

```
acctcon1 -t -l lineuse -o reboots <wtmp | sort
+1n +2 >ctmp

acctcon2 <ctmp | acctmerg >ctacct
```

FILES

/etc/wtmp

USAGE

Administrator.

The command `wtmpfix` [see FWTMP(AS_CMD)] can be used to correct for the confusion caused by date changes.

SEE ALSO

ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

acctmerg — merge or add total accounting files

SYNOPSIS

```
/usr/lib/acct/acctmerg [ options ] [ file ... ]
```

DESCRIPTION

The command `acctmerg` reads its standard input and up to nine additional files, all in the total accounting (**tacct**) format or an ASCII version thereof. It merges these inputs by adding records whose keys (normally user ID and name) are identical, and expects the inputs to be sorted on those keys. `Options` are:

- `-a` Produce output in ASCII version of **tacct**.
- `-i` Input files are in ASCII version of **tacct**.
- `-p` Print input with no processing.
- `-t` Produce a single record that totals all input.
- `-u` Summarize by user ID, rather than user ID and name.
- `-v` Produce output in verbose ASCII format, using more precise notation for floating point numbers.

EXAMPLES

The following sequence is useful for making "repairs" to any file kept in this format:

```
acctmerg -v <file1 >file2
(edit file2 as desired)
acctmerg -i <file2 >file1
```

USAGE

Administrator.

SEE ALSO

ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTCON(AS_CMD), ACCTPRC(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

acctprc1, acctprc2 — process accounting

SYNOPSIS

```
/usr/lib/acct/acctprc1 [ctmp]

/usr/lib/acct/acctprc2
```

DESCRIPTION

The command `acctprc1` reads input in the form produced by ACCT(KE_OS), supplies login names corresponding to user IDs, then writes for each process an ASCII line giving user ID, login name, prime CPU time (tics), non-prime CPU time (tics), and mean memory size (in memory segment units). A memory segment of the mean memory size is a unit of measure for the number of bytes in a logical memory segment on a particular processor. For example, this measure could be in 64-byte units on one machine and in 512-byte units on another. If `ctmp` is given, it is expected to contain a list of login sessions, in the form described in ACCTCON(AS_CMD), sorted by user ID and login name. If this file is not supplied, it obtains login names from the password file. The information in `ctmp` is used to distinguish among different login names that share the same user ID.

The command `acctprc2` reads records in the form written by `acctprc1`, merges and sorts them by user ID and name, then writes them to the standard output as total accounting records.

These commands are typically used as shown below:

```
acctprc1 ctmp </usr/adm/pacct | acctprc2
>ptacct
```

FILES

/etc/passwd

USAGE

Administrator.

SEE ALSO

ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), FWTMP(AS_CMD), RUNACCT(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

clri — clear i-node

SYNOPSIS

```
/etc/clri file-system i-number ...
```

DESCRIPTION

The command `clri` writes zeros on the 64 bytes occupied by the i-node(s) numbered `i-number`. The argment `file-system` must be a special file name referring to a device containing a file system. After `clri` is executed, any blocks in the affected file will show up as "missing" in an `fsck` [see FSCK(AS_CMD)] of the `file-system`. This command should only be used in emergencies and extreme care should be exercised.

Read and write permission is required on the specified `file-system` device. The i-node becomes allocatable.

The primary purpose of this routine is to remove a file which for some reason appears in no directory. If it is used to clear an i-node which does appear in a directory, care should be taken to track down the entry and remove it. Otherwise, when the i-node is reallocated to some new file, the old entry will still point to that file. At that point removing the old entry will destroy the new file. The new entry will again point to an unallocated i-node, so the whole cycle is likely to be repeated again and again.

USAGE

Administrator.

SEE ALSO

FSCK(AS_CMD), FSDB(AS_CMD), NCHECK(AS_CMD).

LEVEL

Level 2: December 1, 1985.

NAME

devnm — device name

SYNOPSIS

```
/etc/devnm [ pathname ]
```

DESCRIPTION

The command `devnm` identifies the special file associated with the mounted file system where the named file or directory resides. The full path name must be given.

EXAMPLE

The command:

```
/etc/devnm /usr
```

produces

```
/dev/dsk/0s1 /usr
```

if `/usr` is mounted on `/dev/dsk/0s1`.

FILES

```
/dev/dsk/*
/etc/mnttab
```

USAGE

Administrator.

SEE ALSO

SETMNT(AS_CMD).

LEVEL

Level 1.

NAME

diskusg, acctdisk — generate disk accounting data by user ID

SYNOPSIS

```
/usr/lib/acct/diskusg [ options ] [ special-file ... ]

/usr/lib/acct/acctdisk
```

DESCRIPTION

The command `diskusg` generates disk accounting information for the file-system identified by the `special-files`. `Diskusg` prints lines on the standard output, one per user, in the following format:

```
uid    login    #blocks
```

where

uid - the numerical user ID of the user.

login - the login name of the user; and

#blocks - the total number of disk blocks allocated to this user.

The command `diskusg` recognizes the following options:

`-s` The input data is already in `diskusg` output format. The command `diskusg` combines all lines for a single user into a single line.

`-v` Verbose; print a list on standard error of all files that are charged to no one.

`-i fnmlist` Ignore the data on those file systems whose file system name is in `fnmlist`. The argument `fnmlist` is a list of file system names separated by commas or enclose within quotes. The command \f(CWdiskusg compares each name in this list with the file system name stored in the volume ID [see `labelit` in VOLCOPY(AS_CMD)].

`-p file` Use `file` as the name of the password file to generate login names. `/etc/passwd` is used by default.

`-u file` Write records to `file` of files that are charged to no one. Records consist of the special file name, the i-node number, and the user ID.

The argument `acctdisk` expects a sequence of disk accounting information, as produced by `diskusg` (sorted by user-id and login name), and generates total accounting records that can be merged with other accounting records. The command `diskusg` is normally run in `dodisk` [see ACCT(AS_CMD)].

FILES

`/etc/passwd`used for user ID to login name conversions

USAGE

Administrator.

SEE ALSO

ACCT(AS_CMD), VOLCOPY(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

fsck — file system consistency check and interactive repair

SYNOPSIS

```
/etc/fsck [ options ] [ file-systems ]
```

DESCRIPTION

The command `fsck` audits and interactively repairs inconsistent conditions for files. If the file system is consistent then the number of files, number of blocks used, and number of blocks free are reported. If the file system is inconsistent the user is prompted for concurrence before each correction is attempted. It should be noted that most corrective actions will result in some loss of data. The amount of data lost and its severity may be determined from the diagnostic output. The default action for each consistency correction is to wait for the user to respond `yes` or `no`. If the user does not have write permission `fsck` will default to a `-n` action.

The file system should be unmounted while `fsck` is used. If this is not possible, care should be taken that the system is quiescent and that it is rebooted immediately afterwards.

The following options are interpreted by `fsck`.

`-y` Assume a yes response to all questions asked by `fsck`.

`-n` Assume a no response to all questions asked by `fsck`; do not open the file system for writing.

`-sX` Ignore the actual free list and (unconditionally) reconstruct a new one. `X` is a hardware dependent option, which specifes how the free list is to be created; if it is not given, the values used when the file system was created, or other default values, are used.

`-SX` Conditionally reconstruct the free list. This option is like `-sX` above except that the free list is rebuilt only if there were no discrepancies discovered in the file system. Using `-S` will force a no response to all questions asked by `fsck`. This option is useful for forcing free list reorganization on uncontaminated file systems.

`-t file` If `fsck` cannot obtain enough memory to keep its tables, it uses a scratch file. If the `-t` option is specified, the file named in the next argument is used as the scratch file, if needed. Without the `-t` flag, `fsck` will prompt the user for the name of the scratch file. The file chosen should not be on the file system being checked, and if it is not a special file or did not already exist, it is removed when `fsck` completes.

`-q` Quiet `fsck`. Do not print size-check messages in Phase 1. Unreferenced FIFOs will silently be removed. If `fsck` requires it, counts in the superblock will be automatically fixed and the free list salvaged.

`-D` Directories are checked for bad blocks. Useful after system crashes.

`-f` Fast check. Check block and sizes (Phase 1) and check the free list (Phase 5). The free list will be reconstructed (Phase 6) if it is necessary.

If no `file-systems` are specified, `fsck` will read a list of default file systems from the file `/etc/checklist`.

Inconsistencies checked are as follows:

1. Blocks claimed by more than one i-node or the free list.
2. Blocks claimed by an i-node or the free list outside the range of the file system.
3. Incorrect link counts.
4. Size checks:
   Incorrect number of blocks.
   Directory size not 16-byte aligned.
5. Bad i-node format.
6. Blocks not accounted for anywhere.
7. Directory checks:
   File pointing to unallocated i-node.
   I-node number out of range.
8. Super Block checks:
   More than {INODE_MAX} inodes.
   More blocks for i-nodes than there are in the file system.
9. Bad free block list format.
10. Total free block and/or free i-node count incorrect.

Orphaned files and directories (allocated but unreferenced) are, with the user's concurrence, reconnected by placing them in the `lost+found` directory, if the files are nonempty. The user will be notified if the file or directory is empty or not. If it is empty, `fsck` will silently remove them. `Fsck` will force the reconnection of nonempty directories. The name assigned is the i-node number. The only restriction is that the directory `lost+found` must preexist in the root of the file system being checked and must have empty slots in which entries can be made. This is accomplished by making `lost+found`, copying a number of files to the directory, and then removing them (before `fsck` is executed).

Checking the raw device is almost always faster and should be used with everything but the `root` file system.

FILES

/etc/checklist

USAGE

Administrator.

I-node numbers for . and .. in each directory should be checked for validity.

LEVEL

Level 1.

NAME

fsdb — file system debugger

SYNOPSIS

```
/etc/fsdb special [ - ]
```

DESCRIPTION

The command `fsdb` can be used to patch up a damaged file system after a crash. It has conversions to translate block and i-numbers into their corresponding disk addresses. Also included are mnemonic offsets to access different parts of an i-node. These greatly simplify the process of correcting control block entries or descending the file system tree.

The command `fsdb` contains several error-checking routines to verify i-node and block addresses. These can be disabled if necessary by invoking `fsdb` with the optional `-` argument or by the use of the `O` symbol. (from the superblock of the file system as the basis for these checks.)

Numbers are considered decimal by default. Octal numbers must be prefixed with a zero. During any assignment operation, numbers are checked for a possible truncation error due to a size mismatch between source and destination.

The command `fsdb` reads a block at a time and will therefore work with raw as well as block I/O. A buffer management routine is used to retain commonly used blocks of data in order to reduce the number of read system calls. All assignment operations result in an immediate write-through of the corresponding block.

The symbols recognized by `fsdb` are:

`#`	absolute address
`i`	convert from i-number to i-node address
`b`	convert to block address
`d`	directory slot offset
`+,-`	address arithmetic
`q`	quit
`>,<`	save, restore an address
`=`	numerical assignment
`=+`	incremental assignment
`=-`	decremental assignment
`="`	character string assignment
`O`	error checking flip flop
`p`	general print facilities
`f`	file print facility
`B`	byte mode
`W`	word mode
`D`	double word mode
`!`	escape to the command interpreter

The print facilities generate a formatted output in various styles. The current address is normalized to an appropriate boundary before printing begins. It advances with the printing and is left at the address of the last item printed. The output can be terminated at any time by typing the delete character. If a number follows the `p` symbol, that many entries are printed. A check is made to detect block boundary overflows since logically sequential blocks are generally not physically sequential. If a count of zero is used, all entries to the end of the current block are printed. The print options available are:

`i`	print as i-nodes
`d`	print as directories
`o`	print as octal words
`e`	print as decimal words
`c`	print as characters
`b`	print as octal bytes

The `f` symbol is used to print data blocks associated with the current i-node. If followed by a number, that block of the file is printed. (Blocks are numbered from zero.) The desired print option letter follows the block number, if present, or the `f` symbol. This print facility works for small as well as large files. It checks for special devices and that the block pointers used to find the data are not zero.

Dots, tabs, and spaces may be used as function delimiters but are not necessary. A line with just a new-line character will increment the current address by the size of the data type last printed. That is, the address is set to the next byte, word, double word, directory entry or i-node, allowing the user to step through a region of a file system. Information is printed in a format appropriate to the data type. Bytes, words and double words are displayed with the octal address followed by the value in octal and decimal. A `.B` or `.D` is appended to the address for byte and double word values, respectively. Directories are printed as a directory slot offset followed by the decimal i-number and the character representation of the entry name. I-nodes are printed with labeled fields describing each element.

The following mnemonics are used for i-node examination and refer to the current working i-node:

`md`	mode
`ln`	link count
`uid`	user ID number
`gid`	group ID number
`sz`	file size
`a#`	data block numbers (0 − 12)
`at`	access time
`mt`	modification time
`maj`	major device number
`min`	minor device number

EXAMPLES

`386i`	prints i-number 386 in an i-node format. This now becomes the current working i-node.
`ln=4`	changes the link count for the working i-node to 4.
`ln=+1`	increments the link count by 1.
`fc`	prints, in ASCII, block zero of the file associated with the working i-node.
`2i.fd`	prints the first 32 directory entries for the root i-node of this file system.
`d5i.fc`	changes the current i-node to that associated with the 5th directory entry (numbered from zero) found from the above command. The first logical block of the file is then printed in ASCII.
`512B.p0o`	prints the superblock of this file system in octal.
`2i.a0b.d7=3`	changes the i-number for the seventh directory slot in the root directory to 3. This example also shows how several operations can be combined on one command line.
`d7.nm=`	changes the name field in the directory slot to the given string. Quotes are optional when used with `nm` if the first character is alphabetic.
`a2b.p0d`	prints the third block of the current i-node as directory entries.

USAGE

Administrator.

SEE ALSO

FSCK(AS_CMD).

LEVEL

Level 1.

NAME

fuser — identify processes using a file or file structure

SYNOPSIS

```
/etc/fuser [-ku] files [-] [[-ku] files]
```

DESCRIPTION

The command `fuser` lists the process IDs of the processes using the `files` specified as arguments. For block special devices, all processes using any file on that device are listed. The process ID is followed by `c`, `p` or `r` if the process is using the file as its current directory, the parent of its current directory (only when in use by the system), or its `root` directory, respectively. If the `-u` option is specified, the login name, in parentheses, also follows the process ID. In addition, if the `-k` option is specified, the SIGKILL signal is sent to each process. Only the super-user can terminate another user's process [see KILL(BA_OS)] Options may be respecified between groups of files. The new set of options replaces the old set, with a lone dash canceling any options currently in force.

The process IDs are printed as a single line on the standard output, separated by spaces and terminated with a single new line. All other output is written on standard error.

EXAMPLES

`fuser -ku /dev/dsk/1s?`
: will terminate all processes that are preventing disk drive one from being unmounted if typed by the super-user, listing the process ID and login name of each as it is killed.

`fuser -u /etc/passwd`
: will list process IDs and login names of processes that have the password file open.

`fuser -ku /dev/dsk/1s? -u /etc/passwd`
: will do both of the above examples in a single command line.

USAGE

Administrator.

The command `fuser` works with a snapshot of the system tables, which is true only for an instant. It is possible that other processes begin accessing the specified file(s) after this snapshot is taken.

SEE ALSO

KILL(BA_OS).

LEVEL

Level 1.

NAME

fwtmp, wtmpfix — manipulate connect accounting records

SYNOPSIS

```
/usr/lib/acct/fwtmp [-ic]

/usr/lib/acct/wtmpfix [files]
```

DESCRIPTION

fwtmp

The command `fwtmp` reads from the standard input and writes to the standard output, converting binary records of the type found in `/etc/wtmp` to formatted ASCII records. The ASCII version is useful to enable editing bad records or general purpose maintenance of the file.

The argument `-ic` is used to denote that input is in ASCII form, and output is to be written in binary form.

wtmpfix

The command `wtmpfix` examines the standard input or named files in `wtmp` format, corrects the time/date stamps to make the entries consistent, and writes to the standard output. A `-` can be used in place of `files` to indicate the standard input. If time/date corrections are not performed, `acctcon1` will fault when it encounters certain date-change records.

Each time the date is set, a pair of date change records are written to `/etc/wtmp`. The first record is the old date denoted by the string `old time` placed in the line field and the flag `OLD_TIME` placed in the type field of the `<utmp.h>` structure. The second record specifies the new date and is denoted by the string `new time` placed in the line field and the flag `NEW_TIME` placed in the type field. The command `wtmpfix` uses these records to synchronize all time stamps in the file.

In addition to correcting time/date stamps, `wtmpfix` will check the validity of the name field to ensure that it consists solely of alphanumeric characters or spaces. If it encounters a name that is considered invalid, it will change the login name to INVALID and write a diagnostic to the standard error. In this way, `wtmpfix` reduces the chance that `acctcon1` will fail when processing connect accounting records.

FILES

`/etc/wtmp`

USAGE

Administrator.

SEE ALSO

ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), RUNACCT(AS_CMD), ACCT(KE_OS).

LEVEL

Level 1.

NAME

init — change system run level

SYNOPSIS

`/etc/init [ 0123456sq ]`

DESCRIPTION

The command `init` is used to direct the actions of the `init` process, which is the system process spawner. (The `init` command provides the `init` process with certain directives; it is important to keep in mind the distinction between the two.)

The system is in a particular `run-level` at any given time. The processes spawned by the `init` process for each of these `run-levels` is defined in the `/etc/inittab` file. The system can be in one of eight `run-levels`, `0–6` and `s` (or `S`). The `run-level` is changed when the System Administrator runs the `init` command.

If the `run-level s` (`S`) is specified, the `init` process goes into the SINGLE-USER level. This is the only `run-level` that does not require the existence of a properly formatted `/etc/inittab` file. (If that file does not exist, then by default the SINGLE USER level is entered.)

If a `run-level` of `0` through `6` is specified, the `init` process enters the corresponding `run-level`.

The following arguments are accepted by `init`:

`0–6` tells `init` to place the system in one of the `run-levels 0–6`.

`q` (or `Q`) tells `init` to re-examine the `/etc/inittab` file. It is often used after that file has been changed, in order to check its correctness.

`s` (or `S`) tells `init` to enter the SINGLE-USER level. When this level change is effected, the virtual system terminal, `/dev/console`, is changed to the terminal from which the command was executed.

FILES

`/etc/inittab`

USAGE

Administrator.

LEVEL

Level 1.

NAME

ipcrm — remove a message queue, semaphore set or shared memory id

SYNOPSIS

`ipcrm [ options ]`

DESCRIPTION

The command `ipcrm` will remove one or more specified message, semaphore or shared memory identifiers. The identifiers are specified by the following `options`:

`-q` *msqid* removes the message queue identifier *msqid* from the system and destroys the message queue and data structure associated with it.

`-m` *shmid* removes the shared memory identifier *shmid* from the system. The shared memory segment and data structure associated with it are destroyed after the last detach operation.

`-s` *semid* removes the semaphore identifier *semid* from the system and destroys the set of semaphores and data structure associated with it.

`-Q` *msgkey* removes the message queue identifier, created with key *msgkey*, from the system and destroys the message queue and data structure associated with it.

`-M` *shmkey* removes the shared memory identifier, created with key *shmkey*, from the system. The shared memory segment and data structure associated with it are destroyed after the last detach.

`-S` *semkey* removes the semaphore identifier, created with key *semkey*, from the system and destroys the set of semaphores and data structure associated with it.

The details of the removes are described in MSGCTL(KE_OS), SHMCTL(KE_OS), and SEMCTL(KE_OS). The identifiers and keys may be found by using IPCS(AS_CMD).

SEE ALSO

IPCS(AS_CMD), MSGCTL(KE_OS), MSGGET(KE_OS), MSGOP(KE_OS), SEMCTL(KE_OS), SEMGET(KE_OS), SEMOP(KE_OS), SHMCTL(KE_OS), SHMGET(KE_OS), SHMOP(KE_OS).

LEVEL

Level 1.

NAME

ipcs – report inter-process communication facilities status

SYNOPSIS

`ipcs [ options ]`

DESCRIPTION

The command `ipcs` prints certain information about active inter-process communication facilities. Without `options`, information is printed in short format for message queues, shared memory, and semaphores that are currently active in the system. Otherwise, the information that is displayed is controlled by the following `options`:

`-q` Print information about active message queues.

`-m` Print information about active shared memory segments.

`-s` Print information about active semaphores.

If any of the options `-q`, `-m`, or `-s` are specified, information about only those indicated will be printed. If none of these three are specified, information about all three will be printed.

`-b` Print biggest allowable size information. (Maximum number of bytes in messages on queue for message queues, size of segments for shared memory, and number of semaphores in each set for semaphores.) See below for meaning of columns in a listing.

`-c` Print creator's login name and group name. See below.

`-o` Print information on outstanding usage. (Number of messages on queue and total number of bytes in messages on queue for message queues and number of processes attached to shared memory segments.)

`-p` Print process number information. (Process ID of last process to send a message, process ID of last process to receive a message on message queues, process ID of creating process, process ID of last process to attach or detach on shared memory segments) See below.

`-t` Print time information. (Time of the last control operation that changed the access permissions for all facilities. Time of last *msgsnd* and last *msgrcv* operations on message queues, last *shmat* and last *shmdt* operations on shared memory, last *semop* operation on semaphores.) See below.

`-a` Use all print `options`. (This is a shorthand notation for `-b`, `-c`, `-o`, `-p`, and `-t`.)

`-C` *corefile*
Use the file `corefile` in place of `/dev/kmem`.

`-N` *namelist*
The argument will be taken as the name of an alternate

`namelist` file, instead of the default.

The column headings and the meaning of the columns in an `ipcs` listing are given below; the letters in parentheses indicate the `options` that cause the corresponding heading to appear; **all** means that the heading always appears. Note that these `options` only determine what information is provided for each facility; they do *not* determine which facilities will be listed.

`T` (all)
Type of the facility:

- `q` message queue;
- `m` shared memory segment;
- `s` semaphore.

ID (all)
The identifier for the facility entry.

KEY (all)
The key used as an argument in calls to `msgget`, `semget`, or `shmget` to create the facility entry. (Note: The key of a shared memory segment is changed to **IPC_PRIVATE** when the segment has been removed until all processes attached to the segment detach it.)

MODE (all)
The facility access modes and flags: The mode consists of 11 characters, interpreted as follows.

The first character is:

- **S** if a process is waiting on a *msgsnd* operation;
- **D** if the associated shared memory segment has been removed. It will disappear when the last process attached to the segment detaches it;
- **–** if the corresponding condition is not true.

The second character is:

- **R** if a process is waiting on a *msgrcv* operation;
- **C** if the associated shared memory segment is to be cleared when the first attach operation is executed;
- **–** if the corresponding condition is not true.

The next 9 characters are interpreted as three sets of three bits each. The first set refers to the owner's permissions; the next to permissions of others in the user-group of the facility entry; and the last to all others. Within each set, the first character indicates permission to read, the second character indicates

permission to write or alter the facility entry, and the last character is currently unused.

The permissions are indicated as follows:

r if read permission is granted;

w if write permission is granted;

a if alter permission is granted;

- if the indicated permission is *not* granted.

(Thus the first character in a set of three can either be **r** or **-**; the second character can be **w**, **a**, or **-**; the last character can only be **-**.)

OWNER (all)
The login name of the owner of the facility entry.

GROUP (all)
The group name of the group of the owner of the facility entry.

CREATOR (a,c)
The login name of the creator of the facility entry.

CGROUP (a,c)
The group name of the group of the creator of the facility entry.

CBYTES (a,o)
The number of bytes in messages currently outstanding on the associated message queue.

QNUM (a,o)
The number of messages currently outstanding on the associated message queue.

QBYTES (a,b)
The maximum number of bytes allowed in messages outstanding on the associated message queue.

LSPID (a,p)
The process ID of the last process to send a message to the associated queue.

LRPID (a,p)
The process ID of the last process to receive a message from the associated queue.

STIME (a,t)
The time the last message was sent to the associated queue.

RTIME (a,t)
The time the last message was received from the associated queue.

CTIME (a,t)
The time when the associated entry was created or changed.

NATTCH (a,o)
The number of processes attached to the associated shared memory segment.

SEGSZ (a,b)
The size of the associated shared memory segment.

CPID (a,p)
The process ID of the creator of the shared memory entry.

LPID (a,p)
The process ID of the last process to attach or detach the shared memory segment.

ATIME (a,t)
The time the last attach was completed to the associated shared memory segment.

DTIME (a,t)
The time the last detach was completed on the associated shared memory segment.

NSEMS (a,b)
The number of semaphores in the set associated with the semaphore entry.

OTIME (a,t)
The time the last semaphore operation was completed on the set associated with the semaphore entry.

SEE ALSO

MSGOP(KE_OS), SEMOP(KE_OS), SHMOP(KE_OS).

USAGE

Things can change while `ipcs` is running; therefore the status it reports may no longer be accurate at the time it is seen.

LEVEL

Level 1.

NAME

killall — kill all active processes

SYNOPSIS

`/etc/killall [ signal ]`

DESCRIPTION

The command `killall` is a procedure used to kill all active processes not directly related to the calling procedure.

The command `killall` is chiefly used to terminate all processes with open files so that the mounted file systems will be unbusied and can be unmounted.

The command `killall` sends `signal` to all remaining processes not belonging to the above group of exclusions. If no `signal` is specified, SIGKILL is used.

USAGE

Administrator.

LEVEL

Level 1.

NAME

link, unlink — exercise link and unlink system calls

SYNOPSIS

```
/etc/link file1 file2

/etc/unlink file
```

DESCRIPTION

The commands `link` and `unlink` perform their respective system calls on their arguments, without any error checking.

These commands may only be executed by the super-user.

USAGE

Administrator.

SEE ALSO

LINK(BA_OS), UNLINK(BA_OS).

LEVEL

Level 1.

NAME

mkfs — construct a file system

SYNOPSIS

```
/etc/mkfs special blocks[:i-nodes] [gap blocks/cyl]

/etc/mkfs special proto [gap blocks/cyl]
```

DESCRIPTION

The command `mkfs` constructs a file system by writing on the special file according to the directions found in the remainder of the command line. The command waits 10 seconds before starting to construct the file system. If the second argument is given as a string of digits, `mkfs` builds a file system with a single empty directory on it. The size of the file system is the value of `blocks` interpreted as a decimal number. This is the number of 512-byte units the file system will occupy. The boot program is left uninitialized.

If the second argument is a file name that can be opened, `mkfs` assumes it to be a prototype file `proto`, and will take its directions from that file. The prototype file contains tokens separated by spaces or new-lines. The first token is the name of a file to be copied onto block zero as the bootstrap program. The second token is a number specifying the size of the created file system in physical disk blocks. Typically it will be the number of blocks on the device, perhaps diminished by space for swapping. The next token is the number of i-nodes in the file system. The next set of tokens comprise the specification for the root file. File specifications consist of tokens giving the mode, the user ID, the group ID, and the initial contents of the file. The syntax of the contents field depends on the mode.

The mode token for a file is a 6-character string. The first character specifies the type of the file. (The characters **−bcd** specify regular, block special, character special and directory files respectively.) The second character of the type is either **u** or **−** to specify set-user-ID mode or not. The third is **g** or **−** for the set-group-ID mode. The rest of the mode is a 3 digit octal number giving the owner, group, and other read, write, execute permissions.

Two decimal number tokens come after the mode; they specify the user and group IDs of the owner of the file.

If the file is a regular file, the next token is a path name whence the contents and size are copied. If the file is a block or character special file, two decimal number tokens follow which give the major and minor device numbers. If the file is a directory, `mkfs` makes the entries and then reads a list of names and (recursively) files specifications for the entries in the directory. The scan is terminated with the token **$**.

If a prototype is used, there is an upper limit on the size of a file that can be initialized. This limit is implementation dependent, but is at least 64K bytes.

A sample prototype specification follows:

```
/stand/diskboot
4872 110
d--755 3 1
usr       d--755 3 1
          sh              ---755 3 1 /bin/sh
          ken             d--755 6 1
                          $
          b0              b--644 3 1 0 0
          c0              c--644 3 1 0 0
          $
$
```

USAGE

Administrator.

LEVEL

Level 1.

NAME

mknod – build special file

SYNOPSIS

```
/etc/mknod name c | b major minor

/etc/mknod name p
```

DESCRIPTION

The command `mknod` makes a directory entry and corresponding i-node for a special file.

The command `mknod` can also be used to create FIFOs (named pipes) (second case in **SYNOPSIS** above).

The first argument is the `name` of the entry. In the first case above, the second argument is `b` if the special file is block-type (disks, tape) or `c` if it is character-type (other devices). The last two arguments are numbers specifying the *major* device type and the *minor* device (e.g., unit, drive, or line number), which may be either decimal or octal (any number with a leading zero).

The assignment of major device numbers is specific to each system.

The command `mknod` may only be used by the superuser, to make special files.

USAGE

Administrator.

SEE ALSO

MKNOD(BA_OS).

LEVEL

Level 1.

NAME

mount, umount — mount and dismount file system

SYNOPSIS

```
/etc/mount [ special directory [ -r ] ]
/etc/umount special
```

DESCRIPTION

The command `mount` announces to the system that a removable file system is present on the device `special`. The `directory` must exist already; it becomes the name of the root of the newly mounted file system.

These commands maintain a table of mounted devices. If invoked with no arguments, `mount` prints the table.

The option `-r` indicates that the file is to be mounted read-only. Physically write-protected and magnetic tape file systems must be mounted in this way or errors will occur when access times are updated, whether or not any explicit write is attempted.

The command `umount` announces to the system that the removable file system previously mounted on device `special` is to be removed.

ERRORS

The command `mount` issues a warning if the file system to be mounted is currently mounted under another name.

The command `umount` reports an error if the special file is not mounted or if it is busy. The file system is busy if it contains an open file, a user's working directory, or another mounted file system.

FILES

`/etc/mnttab` mount table

USAGE

Administrator.

Some degree of validation is done on the file system; however, it is generally unwise to mount garbage file systems.

SEE ALSO

SETMNT(AS_CMD), MOUNT(BA_OS), UMOUNT(BA_OS).

LEVEL

Level 1.

NAME

mvdir — move a directory

SYNOPSIS

```
/etc/mvdir dirname  name
```

DESCRIPTION

The command `mvdir` moves directories within a file system. The argument `dirname` must be a directory; `name` must not be an existing file. If `name` is a directory, then `dirname` is moved to `name/dirname` provided no such file or directory already exists. Neither name may be a sub-set of the other (`/x/y` cannot be moved to `/x/y/z`, nor vice versa).

Only the super-user can use `mvdir`.

USAGE

Administrator.

LEVEL

Level 1.

NAME

ncheck — generate names from i-numbers

SYNOPSIS

```
/etc/ncheck [ -i i-numbers ] [ -a ] [ -s ] [
file-system ]
```

DESCRIPTION

The command `ncheck` with no argument generates a path-name vs. i-number list of all files on a set of default file systems. Names of directory files are followed by `/.`. The `-i` option reduces the report to only those files whose i-numbers follow. The `-a` option allows printing of the names `/.` and `/..`, which are ordinarily suppressed. The `-s` option reduces the report to special files and files with set-user-ID mode; it is intended to discover concealed violations of security policy.

A file system may be specified.

ERRORS

When the file system structure is improper, `??` denotes the "parent" of a parentless file and a path-name beginning with `...` denotes a loop.

USAGE

Administrator.

SEE ALSO

FSCK(AS_CMD).

LEVEL

Level 2: December 1, 1985.

NAME

nice — run a command at low priority

SYNOPSIS

`nice [ -increment ] command`

DESCRIPTION

The command `nice` executes `command` with a lower CPU scheduling priority.

The `increment` is a positive integer less than {NZERO}; if it is not given, the default is half of that (rounded up).

The super-user may run commands with priority higher than normal by using a negative increment, e.g., `nice --2`.

An `increment` larger than the maximum is equivalent to the maximum.

The command `nice` returns the exit status of the subject command.

USAGE

General, except for super-user restriction stated above.

SEE ALSO

NICE(KE_OS).

LEVEL

Level 1.

NAME

pwck, grpck — password/group file checkers

SYNOPSIS

```
/etc/pwck [file]

/etc/grpck [file]
```

DESCRIPTION

The command `pwck` scans the password file and notes any inconsistencies. The checks include validation of the number of fields, login name, user ID, group ID, and whether the login directory and optional program name exist. The default password file is `/etc/passwd`.

The command `grpck` verifies all entries in the group file. This verification includes a check of the number of fields, group name, group ID, and whether all login names appear in the password file. In addition, group entries in `/etc/group` with no login names are flagged. The default group file is `/etc/group`.

FILES

```
/etc/group
/etc/passwd
```

USAGE

Administrator.

LEVEL

Level 1.

NAME

runacct — run daily accounting

SYNOPSIS

`/usr/lib/acct/runacct [mmdd [state]]`

DESCRIPTION

The command `runacct` is the main daily accounting procedure. It is normally initiated via `cron`. The command `runacct` processes connect, fee, disk, and process accounting files. It also prepares summary files for `prdaily` or billing purposes.

Unless otherwise specified, files named here reside in the directory `/usr/adm/acct/nite`.

The command `runacct` takes care not to damage active accounting files or summary files in the event of errors. It records its progress by writing descriptive diagnostic messages into the file `active`. When an error is detected, a message is written to `/dev/console`, mail is sent to the users `root` and `adm`, and `runacct` terminates. The command `runacct` uses a series of lock files to protect against re-invocation. The files `lock` and `lock1` are used to prevent simultaneous invocation, and the file `lastdate` is used to prevent more than one invocation per day.

The command `runacct` breaks its processing into separate, restartable `states` using `statefile` to remember the last `state` completed. It accomplishes this by writing the `state` name into `statefile`. The command `runacct` then looks in `statefile` to see what it has done and to determine what to process next. The `states` are executed in the following order:

SETUP

Move active accounting files into working files.

WTMPFIX

Verify integrity of `/etc/wtmp` file, correcting date changes if necessary.

CONNECT1

Produce connect session records [see ACCTCON(AS_CMD)].

CONNECT2

Convert session records into total accounting records.

PROCESS

Convert process accounting records into total accounting records.

MERGE

Merge the total connect and process accounting records.

FEES

Convert output of `chargefee` into total accounting records and merge with the above (connect and process) total accounting records.

DISK
Merge disk total accounting records with the above (connect, process, and fee) total accounting records. This merge forms the daily total accounting records.

MERGETACCT
Merge the daily total accounting records with the summary total accounting records in `/usr/adm/acct/sum/tacct`.

CMS
Produce command summaries in internal format.

USEREXIT
Any installation-dependent accounting programs can be included here.

CLEANUP
Write ASCII command summaries into the file `/usr/adm/acct/sum/rprtxxxx` [see `prdaily` in ACCT(AS_CMD)]. Remove temporary files and exit.

To restart `runacct` after a failure, first check the `active` file for diagnostics, then fix up any corrupted data files such as `pacct\f(CWor\f(CWwtmp`. The `lock` files and `lastdate` file must be removed before `runacct` can be restarted. The argument `mmdd` is necessary if `runacct` is being restarted, and specifies the month and day for which `runacct` will rerun the accounting. Entry point for processing is based on the contents of `statefile`; to override this, include the desired `state` on the command line to designate where processing should begin.

FILES

```
/usr/src/cmd/acct/nite/active
/usr/src/cmd/acct/nite/lock
/usr/src/cmd/acct/nite/lock1
/usr/src/cmd/acct/nite/lastdate
/usr/src/cmd/acct/nite/statefile
/usr/adm/acct/sum/rprtxxxx
/usr/adm/acct/sum/tacct
/usr/adm/pacct
/etc/wtmp
```

USAGE

Administrator.

Normally `runacct` should not be restarted in the SETUP state. SETUP should be run manually and restart should be done by:

```
runacct mmdd WTMPFIX
```

SEE ALSO

ACCT(AS_CMD), ACCTCMS(AS_CMD), ACCTCOM(AS_CMD), ACCTCON(AS_CMD), ACCTMERG(AS_CMD), ACCTPRC(AS_CMD), CRON(AU_CMD), FWTMP(AS_CMD),

ACCT(KE_OS).

LEVEL

Level 1.

NAME

sa1, sa2, sadc – system activity report package

SYNOPSIS

```
/usr/lib/sa/sadc [t n] [ofile]

/usr/lib/sa/sa1 [t n]

/usr/lib/sa/sa2 [ options ] [-s time] [-e time]
[-i sec]
```

DESCRIPTION

System activity data can be accessed at the special request of a user [see SAR(AS_CMD)] and automatically on a routine basis as described here. The operating system contains a number of counters that are incremented as various system actions occur. These include CPU utilization counters, buffer usage counters, disk and tape I/O activity counters, TTY device activity counters, switching and system-call counters, file-access counters, queue activity counters, and counters for interprocess communications.

The commands `sadc`, `sa1`, and `sa2`, are used to sample, save, and process this data.

The command `sadc`, the data collector, samples system data `n` times every `t` seconds and writes in binary format to `ofile` or to standard output. If `t` and `n` are omitted, a special record is written. This facility is typically used at system boot time to mark the time at which the counters restart from zero.

The utility `sa1`, a variant of `sadc`, is used to collect and store data in the binary file `/usr/adm/sa/sadd` where `dd` is the current day. The options `t` and `n` cause records to be written `n` times at an interval of `t` seconds, (once if the options are omitted).

The utility `sa2`, a variant of `sar`, writes the day's system activity report in the file `/usr/adm/sa/sardd`. The options are explained in SAR(AS_CMD).

FILES

`/usr/adm/sa/sadd` daily data file
`/usr/adm/sa/sardd` daily report file

USAGE

Administrator.

SEE ALSO

SAR(AS_CMD).

LEVEL

Level 1.

NAME

sadp — disk access profiler

SYNOPSIS

```
sadp [ -th ] [ -d device[-drive] ] s [ n ]
```

DESCRIPTION

The command `sadp` reports disk access location and seek distance, in tabular or histogram form. It samples disk activity once every second during an interval of `s` seconds. This is done repeatedly if `n` is specified.

The argument `drive` specifies the disk drives and it may be:

a drive number in the range supported by `device`,

or

two numbers separated by a minus (indicating an inclusive range),

or

a list of drive numbers separated by commas.

The `-d` option may be omitted, if only one `device` is present.

The `-t` option (default) causes the data to be reported in tabular form. The `-h` option produces a histogram of the data on the printer. Default is `-t`.

USAGE

Administrator.

LEVEL

Level 1.

Optional.

NAME

sar — system activity reporter

SYNOPSIS

```
sar [ options ] [-o file] t [ n ]

sar [ options ] [-s time] [-e time] [-i sec]
[-f file]
```

DESCRIPTION

In the first form above, the `sar` command, in the first instance, samples cumulative activity counters in the operating system at `n` intervals of `t` seconds. If the `-o` option is specified, it saves the samples in `file` in binary format. The default value of `n` is 1. In the second form, with no sampling interval specified, `sar` extracts data from a previously recorded `file`, either the one specified by the `-f` option or, by default, the standard system activity daily data file `/usr/adm/sa/sadd` for the current day `dd`. The starting and ending times of the report can be bounded via the `-s` and `-e time` arguments of the form `hh[:mm[:ss]]`. The `-i` option selects records at `sec` second intervals. Otherwise, all intervals found in the data file are reported.

In either case, subsets of data to be printed are specified by option:

`-u` Report CPU utilization (the default):
%usr, %sys, %wio, %idle — portion of time running in user mode, running in system mode, idle with some process waiting for block I/O, and otherwise idle.

`-b` Report buffer activity:
bread/s, bwrit/s — transfers per second of data between system buffers and disk or other block devices;
lread/s, lwrit/s — accesses of system buffers;
%rcache, %wcache — cache hit ratios, e. g., 1 — bread/lread;
pread/s, pwrit/s — transfers via raw (physical) device mechanism.

`-d` Report activity for each block device, e. g., disk or tape drive. When data is displayed, the device specification `dsk-` is generally used to represent a disk drive. The device specification used to represent a tape drive is machine dependent. The activity data reported is:
%busy, avque — portion of time device was busy servicing a transfer request, average number of requests outstanding during that time;
r+w/s, blks/s — number of data transfers from or to device, number of bytes transferred in 512-byte units;
avwait, avserv — average time in ms. that transfer requests wait idly on queue, and average time to be serviced (which for disks includes seek, rotational latency and data transfer times).

`-y` Report TTY device activity:
rawch/s, canch/s, outch/s — input character rate, input character rate processed by canon, output character rate;

rcvin/s, xmtin/s, mdmin/s – receive, transmit and modem interrupt rates.

`-c` Report system calls:
scall/s – system calls of all types;
sread/s, swrit/s, fork/s, exec/s – specific system calls;
rchar/s, wchar/s – characters transferred by read and write system calls.

`-w` Report system swapping and switching activity:
swpin/s, swpot/s, bswin/s, bswot/s – number of transfers and number of 512-byte units transferred for swapins and swapouts (including initial loading of some programs);
pswch/s – process switches.

`-a` Report use of file access system routines:
iget/s, namei/s, dirblk/s.

`-q` Report average queue length while occupied, and % of time occupied:
runq-sz, %runocc – run queue of processes in memory and runnable;
swpq-sz, %swpocc – swap queue of processes swapped out but ready to run.

`-v` Report status of process, i-node, file, record lock and file header tables:
proc-sz, inod-sz, file-sz, lock-sz, fhdr-sz – entries/size for each table, evaluated once at sampling point;
ov – overflows that occur between sampling points for each table.

`-m` Report message and semaphore activities:
msg/s, sema/s – primitives per second.

`-A` Report all data (all options effective).

EXAMPLES

To see today's CPU activity so far:

```
sar
```

To watch CPU activity evolve for 10 minutes and save data:

```
sar -o temp 60 10
```

To later review disk and tape activity from that period:

```
sar -d -f temp
```

FILES

`/usr/adm/sa/sadd` daily data file, where `dd` are digits representing the day of the month.

USAGE

Administrator.

SEE ALSO

SA1(AS_CMD).

LEVEL

Level 1.

NAME

setmnt — establish mount table

SYNOPSIS

`/etc/setmnt`

DESCRIPTION

The command `setmnt` creates the `/etc/mnttab` table, which is needed for both the `mount` and `umount` [see MOUNT(AS_CMD)] commands. The command `setmnt` reads standard input and creates a `mnttab` entry for each line. Input lines have the format:

```
filesys node
```

where `filesys` is the name of the file system's *special file* and `node` is the root name of that file system. Thus `filesys` and `node` become the first two strings in the `/etc/mnttab` entry.

FILES

`/etc/mnttab`

USAGE

Administrator.

SEE ALSO

MOUNT(AS_CMD).

LEVEL

Level 1.

NAME

sync — flush system buffers

SYNOPSIS

`sync`

DESCRIPTION

The command `sync` executes the `sync` system source routine. If the system is to be stopped, `sync` must be executed to ensure file system integrity. It will flush all previously unwritten system buffers out to disk, thus assuring that all file modifications up to that point will be saved.

USAGE

Administrator.

SEE ALSO

SYNC(BA_OS).

LEVEL

Level 1.

NAME

sysdef — system definition

SYNOPSIS

```
/etc/sysdef [ opsys [ master ] ]
```

DESCRIPTION

The command `sysdef` analyzes the named operating system file (or the default one if none is specified) and extracts configuration information. The master file contains the hardware and software specifications. (The default master file is used if one is not specified.) This includes all hardware devices as well as system devices and all tunable parameters.

USAGE

Administrator.

LEVEL

Level 1.

NAME

timex — time a command; report process data and system activity

SYNOPSIS

```
timex [options] command
```

DESCRIPTION

The given `command` is executed; the elapsed time, user time and system time spent in execution are reported in seconds. Optionally, process accounting data for the `command` and all its children can be listed or summarized, and total system activity during the execution interval can be reported.

The output of `timex` is written on standard error.

The options are:

`-p` List process accounting records for `command` and all its children. Suboptions `f`, `h`, `k`, `m`, `r`, and `t` modify the data items reported, as defined in ACCTCOM(AS_CMD). The number of blocks read or written and the number of characters transferred are always reported.

`-o` Report the total number of blocks read or written and total characters transferred by `command` and all its children.

`-s` Report total system activity (not just that due to `command`) that occurred during the execution interval of `command`. All the data items listed in SAR(AS_CMD) are reported.

USAGE

General.

SEE ALSO

ACCTCOM(AS_CMD), SAR(AS_CMD).

LEVEL

Level 1.

NAME

volcopy, labelit — copy file systems with label checking

SYNOPSIS

```
/etc/volcopy [options] fsname special1 volname1
special2 volname2

/etc/labelit special [ fsname volume [ -n ] ]
```

DESCRIPTION

volcopy

The command `volcopy` makes a literal copy of the file system using a blocksize matched to the device. The options are:

`-a` invoke a verification sequence requiring a positive operator response instead of the standard delay before the copy is made

`-s` (default) invoke the DEL if wrong verification sequence.

Other options are used only with tapes:

`-bpi`*density*	bits-per-inch
`-feet`*size*	size of reel in feet
`-reel`*num*	beginning reel number for a restarted copy,
`-buf`	use double buffered I/O.

The program requests length and density information if it is not given on the command line or is not recorded on an input tape label. If the file system is too large to fit on one reel, `volcopy` will prompt for additional reels. Labels of all reels are checked. Tapes may be mounted alternately on two or more drives. If `volcopy` is interrupted, it will ask if the user wants to quit or wants to escape to the command interpreter. In the latter case, the user can perform other operations (e.g.: `labelit`) and return to `volcopy` by exiting the command interpreter.

The `fsname` argument represents the file system name on the device (e.g.: `root`, `u1`) being copied.

The `special` should be the physical disk section or tape (e.g.: `/dev/rdsk/1s5`, `/dev/rmt/0m`, etc.).

The `volname` is the physical volume name and should match the external label sticker. Such label names are limited to six or fewer characters. The argument `volname` may be `-` to use the existing volume name.

The arguments `special1` and `volname1` are the device and volume from which the copy of the file system is being extracted. The arguments `special2` and `volname2` are the target device and volume.

labelit

The command `labelit` can be used to provide initial labels for unmounted disk or tape file systems. With the optional arguments omitted, `labelit` prints current label values. The `-n` option provides for initial labeling of new tapes only (this destroys previous contents).

USAGE

Administrator.

LEVEL

Level 1.

NAME

whodo — who is doing what

SYNOPSIS

`/etc/whodo`

DESCRIPTION

The command `whodo` produces merged, reformatted, and dated output from the WHO(AU_CMD) and PS(BU_CMD) commands.

FILES

`/etc/passwd`

USAGE

General.

SEE ALSO

PS(BU_CMD), WHO(AU_CMD).

LEVEL

Level 1.

Part V

Software Development Extension

Chapter 9
Introduction

9.1 OVERVIEW

The Software Development Extension provides facilities for the compilation and maintenance of C language software. Principal components are the C compiler `cc` and its related utilities, the program development aids `yacc` and `lex`, and the Source Code Control System (SCCS) utilities.

The System V Base, the Basic Utilities Extension, and the Advanced Utilities Extension are prerequisites for the Software Development Extension.

9.2 DESCRIPTION

UTILITIES

admin	dis #	make	strip
as #	env	nm	time
cc	get	prof	tsort
cflow	ld	prs	unget
chroot	lex	rmdel	val
cpp	lint	sact	what
cxref	lorder	sdb	xargs
delta	m4	size	yacc

ROUTINES

The following routines were defined in Volume 1. They must be present (that is, compilation of programs that use these routines must be supported) to satisfy the requirements of the Software Development Extension. (See the "C Libraries" section below for a discussion of library specification.)

Operating System Service Routines

abort	fileno	getuid	setpgrp
access	fopen	ioctl	setuid
alarm	fread	kill	signal
calloc	free	link	sleep
chdir	freopen	lockf ++	stat
chmod	fseek	mallinfo +	stime
chown	fstat	malloc	system
clearerr	ftell	mallopt +	time
dup	fwrite	mknod	times
exit	getcwd	pause	ulimit
fclose	getegid	pclose	umask
fcntl	geteuid	pipe	uname
fdopen	getgid	popen	unlink
feof	getpgrp	realloc	ustat
ferror	getpid	rewind	utime
fflush	getppid	setgid	wait
close	execlp	fork	read
creat	execv	lseek	umount
execl	execve	mount	write
execle	execvp	open	
_exit	sync		

General Routines

_tolower	hsearch	memchr	strcpy
_toupper	isalnum	memcmp	strcspn
abs	isalpha	memcpy	strlen
advance	isascii	memset	strncat
asctime	isatty	mktemp	strncmp
atof	iscntrl	modf	strncpy
atoi	isdigit	mrand48	strpbrk
atol	isgraph	nrand48	strrchr
bsearch	islower	perror *	strspn
clock	isprint	putenv +	strtod +
compile	ispunct	qsort	strtok
crypt	isspace	rand	strtol
drand48	isupper	seed48	swab
encrypt	isxdigit	setjmp	tdelete
erand48	jrand48	setkey	tfind +
frexp	lcong48	srand	toascii
ftw	ldexp	srand48	tolower
getenv	lfind +	ssignal **	toupper

getopt	localtime	step	tsearch
gmtime	longjmp	strcat	ttyname
gsignal **	lrand48	strchr	twalk
hcreate	lsearch	strcmp	tzset
hdestroy	memccpy		

Stdio Routines

ctermid	getc	puts	tmpfile
fgetc	getchar	putw	tmpnam
fgets	gets	scanf	ungetc
fprintf	getw	setbuf	vfprintf +
fputc	printf	setvbuf +	vprintf +
fputs	putc	sprintf	vsprintf +
fscanf	putchar	tempnam	

Math Routines

acos	erfc	j1	sinh
asin	exp	jn	sqrt
atan	fabs	log	tan
atan2	floor	log10	tanh
ceil	fmod	matherr	y0
cos	gamma	pow	y1
cosh	hypot	sin	yn
erf	j0		

In addition to the above, the following library routines must also be present. These routines are defined in this volume.

Additional Routines

a64l	getgrgid	getutid	pututline
assert	getgrnam	getutline	setgrent
endgrent	getlogin	l64a	setpwent
endpwent	getpass	MARK #	setutent
endutent	getpwent	monitor #	sgetl
fgetgrent	getpwnam	nlist	sputl
fgetpwent	getpwuid	putpwent	utmpname

getgrent getutent

* Level 2: January 1, 1985.
** Level 2: December 1, 1985.
\# Optional.
\+ New in System V Release 2.

9.3 C LIBRARIES

The standard C library is automatically included (that is, searched to resolve undefined external references). This includes all of the "Operating System Service Routines", "General Routines", and the "Stdio Routines" listed above. Inclusion of other libraries requires specific loader options on the C compiler `cc` command line. For example, the mathematical library ("Math Routines" above) is searched by including the option `-lm` on the command line:

```
cc file.c -lm
```

Notes on other libraries:

- The `lex` library (`cc` option `-ll`) is required for the compilation of programs generated by *lex* [see LEX(SD_CMD)].
- The object file library (`cc` option `-lld`) contains routines used for the manipulation of object files. The only such routines required for this Extension are `sputl` and `sgetl` [see SPUTL(SD_LIB)].
- The `yacc` library (`cc` option `-ly`) facilitates the use of `yacc` [see YACC(SD_CMD)].

Chapter 10
Library Routines

NAME

a64l, l64a — convert between long integer and base-64 ASCII string

SYNOPSIS

```
long a64l (s)
char *s;

char *l64a (l)
long l;
```

DESCRIPTION

These routines are used to maintain numbers stored in *base-64* ASCII characters. This is a notation by which long integers can be represented by up to six characters; each character represents a "digit" in a radix-64 notation.

The characters used to represent 'digits' are **.** for 0, **/** for 1, **0** through **9** for 2—11, **A** through **Z** for 12—37, and **a** through **z** for 38—63.

The routine `a64l` takes a pointer to a null-terminated base-64 representation and returns a corresponding `long` value. If the string pointed to by `s` contains more than six characters, `a64l` will use the first six.

The routine `l64a` takes a `long` argument and returns a pointer to the corresponding base-64 representation. If the argument is 0, `l64a` returns a pointer to a null string.

APPLICATION USAGE

The value returned by `l64a` may be a pointer into a static buffer, the contents of which would therefore be overwritten by each call.

LEVEL

Level 1.

NAME

assert – verify program assertion

SYNOPSIS

```
#include <assert.h>

void assert (expression)
int expression;
```

DESCRIPTION

The `assert` macro is useful for putting diagnostics into programs. When it is executed, if `expression` is false (zero), `assert` prints

```
assertion failed: expression, file xyz, line
nnn
```

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

SEE ALSO

ABORT(BA_OS).

APPLICATION USAGE

Compiling with the preprocessor option `-DNDEBUG` or with the preprocessor control statement `#define NDEBUG` ahead of the `#include <assert.h>` statement will stop assertions from being compiled into the program.

LEVEL

Level 1.

NAME

getgrent, getgrgid, getgrnam, setgrent, endgrent, fgetgrent — get group file entry

SYNOPSIS

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

struct group *getgrent ( )

struct group *getgrgid (gid)
int gid;

struct group *getgrnam (name)
char *name;

void setgrent ( )

void endgrent ( )

struct group *fgetgrent (f)
FILE *f;
```

DESCRIPTION

The routines `getgrent`, `getgrgid` and `getgrnam` each return pointers to an object with the following structure containing the broken-out fields of a line in the `/etc/group` file. Each line contains a 'group' structure, defined in the `<grp.h>` header file.

The structure contains at least the following members:

```
char   *gr_name;   /* the name of the group */
int    gr_gid;     /* the numerical group ID */
char   **gr_mem;   /* array of pointers to member names */
```

The routine `getgrent` when first called returns a pointer to the first group structure in the file; thereafter, it returns a pointer to the next group structure in the file; so, successive calls may be used to search the entire file.

The routine `getgrgid` searches the file until a numerical group id matching `gid` is found, and returns a pointer to the particular structure in which it was found.

The routine `getgrnam` searches the file until a group name matching *name* is found, and returns a pointer to the particular structure in which it was found.

If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

The routine `setgrent` effectively rewinds the group file to allow repeated searches.

The routine `endgrent` may be called to close the group file when processing is complete.

The routine `fgetgrent` returns a pointer to the next group structure in the file `f`; this file must have the format of `/etc/group`.

RETURN VALUE

A NULL pointer is returned on end of file or an error.

FILES

`/etc/group`

SEE ALSO

GETLOGIN(SD_LIB), GETPWENT(SD_LIB).

APPLICATION USAGE

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

LEVEL

Level 1.

NAME

getlogin — get login name

SYNOPSIS

```
char *getlogin ( );
```

DESCRIPTION

The routine `getlogin` returns a pointer to the login name as found in the file `/etc/utmp`. It may be used in conjunction with the routine `getpwnam` [see GETPWENT(SD_LIB)] to locate the correct password file entry when the same user ID is shared by several login names.

If `getlogin` is called within a process that is not attached to a terminal, it returns a NULL pointer. The correct procedure for determining the login name is to call `getlogin` and if it fails to call `getpwuid`.

RETURN VALUE

Returns a NULL pointer if `name` is not found.

FILES

`/etc/utmp`

SEE ALSO

GETGRENT(SD_LIB), GETPWENT(SD_LIB).

APPLICATION USAGE

The return value may point to static data whose content would therefore be overwritten by each call.

LEVEL

Level 1.

NAME

getpass — read a password

SYNOPSIS

```
char *getpass (prompt)
char *prompt;
```

DESCRIPTION

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

FILES

`/dev/tty`

APPLICATION USAGE

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

LEVEL

Level 1.

NAME

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

SYNOPSIS

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

struct passwd *getpwent ( )

struct passwd *getpwuid (uid)
int uid;

struct passwd *getpwnam (name)
char *name;

void setpwent ( )

void endpwent ( )

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

DESCRIPTION

Each of the routines `getpwent`, `getpwuid` and `getpwnam` returns a pointer to a structure containing the broken-out fields of a line in the `/etc/passwd` file. Each line in the file contains a 'passwd' structure, declared in the `<pwd.h>` header file.

The structure contains at least the following members:

```
char   *pw_name;     /* login  name */
char   *pw_passwd;   /* encrypted password */
int    pw_uid;       /* numerical user id */
int    pw_gid;       /* numerical group id */
char   *pw_dir;      /* initial working directory */
char   *pw_shell;    /* command interpreter */
```

The routine `getpwent` when first called returns a pointer to the first passwd structure in the file; thereafter, it returns a pointer to the next passwd structure in the file; so successive calls can be used to search the entire file.

The routine `getpwuid` searches the file until a numerical user ID matching `uid` is found, and returns a pointer to the particular structure in which it was found.

The routine `getpwnam` searches the file until a login name matching `name` is found, and returns a pointer to the particular structure in which it was found.

If an end-of-file or an error is encountered on reading, these functions return a NULL pointer.

The routine `setpwent` effectively of rewinds the password file to allow repeated searches.

The routine `endpwent` may be called to close the password file when processing is complete.

The routine `fgetpwent` returns a pointer to the next passwd structure in the file `f`, which must have the format of `/etc/passwd`.

RETURN VALUE

A NULL pointer is returned on end of file or error.

FILES

`/etc/passwd`

SEE ALSO

GETLOGIN(SD_LIB), GETGRENT(SD_LIB).

APPLICATION USAGE

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

LEVEL

Level 1.

NAME

getutent, getutid, getutline, pututline, setutent, endutent, utmpname — access utmp file entry

SYNOPSIS

```
#include <utmp.h>

struct utmp *getutent ( )

struct utmp *getutid (id)
struct utmp *id;

struct utmp *getutline (line)
struct utmp *line;

void pututline (utmp)
struct utmp *utmp;

void setutent ( )

void endutent ( )

void utmpname (file)
char *file;
```

DESCRIPTION

Each of the routines `getutent`, `getutid` and `getutline` returns a pointer to a structure, which is defined in the header file `<utmp.h>`. The structure contains at least the following members:

```
char    ut_user[];   /* User login name */
char    ut_id[];     /* /etc/inittab id */
char    ut_line[];   /* Device name */
short   ut_pid;      /* Process id */
short   ut_type;     /* Type of entry */
```

In addition, (at least) the following type values for `ut_type` are defined:
`EMPTY, RUN_LVL, BOOT_TIME, OLD_TIME, NEW_TIME, INIT_PROCESS, LOGIN_PROCESS, USER_PROCESS, DEAD_PROCESS, ACCOUNTING.`

The routine `getutent` reads in the next entry from the `/etc/utmp` file. If the file is not already open, it opens it. If it reaches the end of the file, it fails.

The routine `getutid` searches forward from the current point in the `/etc/utmp` file; if the `ut_type` value of the structure `id` is `RUN_LVL`, `BOOT_TIME`, `OLD_TIME` or `NEW_TIME`, then it stops when it finds an entry with a `ut_type` matching the `ut_type` of the structure ID. If the `ut_type` value is `INIT_PROCESS`, `LOGIN_PROCESS`, `USER_PROCESS`, or `DEAD_PROCESS`, then it stops when it finds an entry whose type is one of these four and whose `ut_id` field matches the `ut_id` field of `id`. If the end of file is reached without a match, `getutid` fails.

The routine `getutline` searches forward from the current point in the `/etc/utmp` file until it finds an entry of the type `LOGIN_PROCESS` or `USER_PROCESS` which also has a `ut_line` value matching that of `line`. If the end of file is reached without a match, `getutline` fails.

The routine `pututline` writes out the supplied `/etc/utmp` structure into the `/etc/utmp` file. It uses `getutid` to search forward for the proper place if it finds that it is not already at the proper place. It is expected that normally the user of `pututline` will have searched for the proper entry using one of the above routines. If so, `pututline` will not search. If `pututline` does not find a matching slot for the new entry, it will add a new entry to the end of the file.

The routine `setutent` resets the input stream to the beginning of the file. This should be done before each search for a new entry if it is desired that the entire file be examined.

The routine `endutent` closes the currently open file.

The routine `utmpname` allows the user to change the name of the file examined by these routines, from `/etc/utmp` to any other file. Usually, this other file will be `/etc/wtmp`. If the file does not exist, this will not be apparent until the first attempt to reference the file. (This is because `utmpname` does not open the file. It just closes the old file if it is currently open and saves the new file name.)

RETURN VALUE

A NULL pointer is returned upon failure to read, whether for permissions or having reached the end of file, or upon failure to write.

FILES

`/etc/utmp`
`/etc/wtmp`

APPLICATION USAGE

The most current entry is saved in a static structure. Multiple accesses require that it be copied before further accesses are made.

Each call to either `getutid` or `getutline` sees the routine examine the static structure before performing more I/O. If the contents of the static structure match what it is searching for, it looks no further. For this reason, to use `getutline` to search for multiple occurrences, it would be necessary to zero out the static after each success, or `getutline` would just return the same pointer over and over again.

There is one exception to the rule about removing the structure before further reads are done. The implicit read done by `pututline` (if it finds that it is not already at the correct place in the file) will not hurt the contents of the static structure returned by the `getutent`, `getutid`, or `getutline` routines, if the user has just modified those contents and passed the pointer back to `pututline`.

The sizes of the arrays in the structure can be found using the `sizeof` operator.

LEVEL

Level 1.

NAME

MARK — profile within a function

SYNOPSIS

```
#define MARK
#include <prof.h>

void MARK (name)
```

DESCRIPTION

The macro `MARK` will introduce a mark called `name` that will be treated the same as a function entry point. Execution of the mark will add to a counter for that mark, and program-counter time spent will be accounted to the immediately preceding mark or to the function if there are no preceding marks within the active function.

The identifier `name` may be any combination of letters, numbers or underscores. Each `name` in a single compilation must be unique, but may be the same as any ordinary program symbol.

For marks to be effective, the symbol `MARK` must be defined before the header file `<prof.h>` is included. This may be defined by a preprocessor directive as in the synopsis, or by a command line argument, i.e:

```
cc -p -DMARK foo.c
```

If `MARK` is not defined, the `MARK(name)` statements may be left in the source files containing them and will be ignored.

EXAMPLE

In this example, marks can be used to determine how much time is spent in each loop. Unless this example is compiled with `MARK` defined on the command line, the marks are ignored.

```
#include <prof.h>

foo( )
{
    int i, j;
    .
    .
    .
    MARK(loop1);
    for (i = 0; i < 2000; i++) {
        . . .
    }
    MARK(loop2);
    for (j = 0; j < 2000; j++) {
        . . .
    }
}
```

SEE ALSO

PROFIL(KE_OS), MONITOR(SD_LIB), PROF(SD_CMD).

LEVEL

Level 1.

Optional. (Requires the `profil` system service routine).

NAME

monitor — prepare execution profile

SYNOPSIS

```
#include <mon.h>

void monitor (lowpc, highpc, buffer, bufsize, nfunc)
int (*lowpc)( ), (*highpc)( );
WORD *buffer;
int bufsize, nfunc;
```

DESCRIPTION

The routine `monitor` is an interface to the `profil` system service routine [see PROFIL(KE_OS)]; `lowpc` and `highpc` are the addresses of two functions; `buffer` is the address of a (user supplied) array of `bufsize` WORDs (`WORD` is defined in the `<mon.h>` header file). The `monitor` routine arranges to record a histogram of periodically sampled values of the program counter, and of counts of calls of certain functions, in the buffer. The lowest address sampled is that of `lowpc` and the highest is just below `highpc`; `lowpc` may not equal 0 for this use of `monitor`. At most `nfunc` call counts can be kept; only calls of functions compiled with the profiling option `-p` of `cc` are recorded.

An executable program created by using the `-p` option with `cc` automatically includes calls for the `monitor` routine with default parameters; therefore `monitor` need not be called explicitly except to gain fine control over profiling.

For the results to be significant, especially where there are small, heavily used routines, it is suggested that the buffer be no more than a few times smaller than the range of locations sampled.

To profile the entire program, it is sufficient to use

```
extern int etext();
. . .
monitor ((int (*)())2, etext, buf,
bufsize, nfunc);
```

The routine `etext` lies just above all the program text.

To stop execution monitoring and write the results, use

```
monitor ((int (*)())0, (int (*)())0,
0, 0, 0);
```

The `prof` command [see PROF(SD_CMD)] can then be used to examine the results.

The name of the file written by `monitor` is controlled by the environmental variable `profdir`. If `PROFDIR` is not set, then the file '`mon.out`' is created in the current directory. If `PROFDIR` is set to the null string, then no profiling is done and no output file is created. Otherwise, the value

of `PROFDIR` is used as the name of the directory in which to create the output file. If `PROFDIR` is `dirname`, then the output file is named '`dirname/`*pid*`.mon.out`', where *pid* is the program's process ID. (When `monitor` is called automatically by using the `-p` option of `cc`, the file created is '`dirname/`*pid.progname*', where *progname* is the name of the program.)

FILES

`mon.out`

SEE ALSO

PROFIL(KE_OS), CC(SD_CMD), PROF(SD_CMD).

LEVEL

Level 1.

Optional. (Requires the `profil` system service routine.)

NAME

nlist – get entries from name list

SYNOPSIS

```
#include <nlist.h>

int nlist (file-name, nl)
char *file-name;
struct nlist *nl;
```

DESCRIPTION

The routine `nlist` examines the name list in the executable file whose name is pointed to by *file-name*, and selectively extracts a list of values and puts them in the array of nlist structures pointed to by `nl`. The name list `nl` consists of an array of structures containing names of variables, types and values. The list is terminated with a null name; that is, a null string is in the name position of the structure. Each variable name is looked up in the name list of the file. If the name is found, the type and value of the name are inserted in the next two fields. The type field will be set to 0 unless the file was compiled with the `-g` option of `cc`. If the name is not found, both entries are set to 0.

This function is useful for examining the system name list kept in the namelist file. In this way programs can obtain system addresses that are up to date.

RETURN VALUE

Returns −1 upon error; otherwise returns 0.

All value entries are set to 0 if the file cannot be read or if it does not contain a valid name list.

LEVEL

Level 1.

NAME

putpwent — write password file entry

SYNOPSIS

```
#include <pwd.h>

int putpwent (p, f)
struct passwd *p;
FILE *f;
```

DESCRIPTION

The routine `putpwent` is the inverse of `getpwent`. Given a pointer to a password structure created by `getpwent` (or `getpwuid` or `getpwnam`), `putpwent` writes a line on the file `f`, which must have the format of `/etc/passwd`.

RETURN VALUE

Returns a non-zero value if an error was detected during its operation, otherwise returns 0.

SEE ALSO

GETPWENT(SD_LIB).

LEVEL

Level 1.

NAME

sputl, sgetl — access long integer data in a machine-independent fashion.

SYNOPSIS

```
void sputl (value, buffer)
long value;
char *buffer;

long sgetl (buffer)
char *buffer;
```

DESCRIPTION

The routine `sputl` takes the four bytes of the long integer `value` and places them in memory starting at the address pointed to by `buffer`. The ordering of the bytes is the same across all machines.

The routine `sgetl` retrieves the four bytes in memory starting at the address pointed to by `buffer` and returns the long integer value in the byte ordering of the host machine.

The combination of `sputl` and `sgetl` provides a machine-independent way of storing long numeric data in a file in binary form without conversion to characters.

A program which uses these functions must be compiled with the object file library, by using the `-lld` option of `cc`.

LEVEL

Level 1.

Chapter 11
Commands and Utilities

NAME

admin — create and administer SCCS files

SYNOPSIS

```
admin [ -n ] [ -i[name] ] [ -rrel ] [ -t[name] ]
[ -fflag[flag-val] ] [ -dflag[flag-val] ] [ -alo-
gin ] [ -elogin ] [ -m[mrlist] ] [ -%[comment] ]
[ -h ] [ -z ] file ...
```

DESCRIPTION

The command `admin` is used to create new SCCS files and change parameters of existing ones. Arguments to `admin`, which may appear in any order, consist of options, which begin with `-`, and named files (note that SCCS file names must begin with `s.`). If a named file does not exist, it is created, and its parameters are initialized according to the specified options. Parameters not initialized by an option are assigned a default value. If a named file does exist, parameters corresponding to specified options are changed, and other parameters are left as is.

If a directory is named, `admin` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with `s.`) and unreadable files are silently ignored. If a name of `-` is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The options are as follows. Each is explained as though only one named file is to be processed since the effects of the options apply independently to each named file.

`-n` This option indicates that a new SCCS file is to be created.

`-i`[*name*] The *name* of a file from which the text for a new SCCS file is to be taken. The text constitutes the first delta of the file (see `-r` option for delta numbering scheme). If the `i` option is used, but the file name is omitted, the text is obtained by reading the standard input until an end-of-file is encountered. If this option is omitted, then the SCCS file is created empty. Only one SCCS file may be created by an `admin` command on which the `i` option is supplied. Using a single `admin` to create two or more SCCS files requires that they be created empty (no `-i` option). Note that the `-i` option implies the `-n` option.

`-r`*rel* The *rel*ease into which the initial delta is inserted. This option may be used only if the `-i` option is also used. If the `-r` option is not used, the initial delta is inserted into release 1. The level of the initial delta is always 1 (by default initial deltas are named 1.1).

-t[*name*] The *name* of a file from which descriptive text for the SCCS file is to be taken. If the **-t** option is used and **admin** is creating a new SCCS file (the **-n** and/or **-i** options also used), the descriptive text file name must also be supplied. In the case of existing SCCS files: (1) a **-t** option without a file name causes removal of descriptive text (if any) currently in the SCCS file, and (2) a **-t** option with a file name causes text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file. This option specifies a *flag*, and, possibly, a value for the *flag*, to be placed in the SCCS file. Several **f** options may be supplied on a single **admin** command line. The allowable *flag*s and their values are:

b Allows use of the **-b** option on a **get** command to create branch deltas.

c*ceil* The highest release (i.e., "ceiling"), a number less than or equal to 9999, which may be retrieved by a **get** command for editing. The default value for an unspecified **c** flag is 9999.

f*floor* The lowest release (i.e., "floor"), a number greater than 0 but less than 9999, which may be retrieved by a **get** command for editing. The default value for an unspecified **f** flag is 1.

d*SID* The default delta number (SID) to be used by a **get** command.

i[*str*] Causes the "No id keywords" message issued by **get** or **delta** to be treated as a fatal error. In the absence of this flag, the message is only a warning. The message is issued if no SCCS identification keywords (see **get**) are found in the text retrieved or stored in the SCCS file.

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

l*list* A *list* of releases to which deltas can no longer be made (**get -e** against one of these "locked" releases fails). The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= *RELEASE NUMBER* | **a**

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

n Causes **delta** to create a "null" delta in each of those releases (if any) being skipped when a delta is

made in a *new* release (e.g., in making delta 5.1 after delta 2.7, releases 3 and 4 are skipped). These null deltas serve as "anchor points" so that branch deltas may later be created from them. The absence of this flag causes skipped releases to be non-existent in the SCCS file, preventing branch deltas from being created from them in the future.

q*text* — User definable text substituted for all occurrences of the %Q% keyword in SCCS file text retrieved by `get`.

m*mod* — *Mod*ule name of the SCCS file substituted for all occurrences of the %M% keyword in SCCS file text retrieved by `get`. If the **m** flag is not specified, the value assigned is the name of the SCCS file with the leading **s.** removed.

t*type* — *Type* of module in the SCCS file substituted for all occurrences of %Y% keyword in SCCS file text retrieved by `get`.

v[*pgm*] — Causes `delta` to prompt for Modification Request (*MR*) numbers as the reason for creating a delta. The optional value specifies the name of an *MR* number validity checking program. (If this flag is set when creating an SCCS file, the `m` option must also be used even if its value is null).

-d*flag* — Causes removal (deletion) of the specified *flag* from an SCCS file. The `-d` option may be specified only when processing existing SCCS files. Several `-d` options may be supplied on a single `admin` command. See the `-f` option for allowable *flag* names. (The **l***list* flag gives a *list* of releases to be "unlocked". See the `-f` option for further description of the **l** flag and the syntax of a *list*.)

-a*login* — A *login* name, or numerical group ID, to be added to the list of users which may make deltas (changes) to the SCCS file. A group ID is equivalent to specifying all *login* names common to that group ID. Several `a` options may be used on a single `admin` command line. As many *login*s, or numerical group IDs, as desired may be on the list simultaneously. If the list of users is empty, then anyone may add deltas. If *login* or group ID is preceded by a `!` they are to be denied permission to make deltas.

-e*login* — A *login* name, or numerical group ID, to be erased from the list of users allowed to make deltas (changes) to the SCCS file. Specifying a group ID is equivalent to specifying all *login* names common to that group ID. Several `e` options

may be used on a single `admin` command line.

-y[*comment*] The *comment* text is inserted into the SCCS file as a comment for the initial delta in a manner identical to that of `delta`. Omission of the `-y` option results in a default comment line being inserted in the form:
date and time created *YY/MM/DD HH:MM:SS* by *login*
The `-y` option is valid only if the `-i` and/or `-n` options are specified (i.e., a new SCCS file is being created).

-m[*mrlist*] The list of Modification Requests (*MR*) numbers is inserted into the SCCS file as the reason for creating the initial delta in a manner identical to `delta`. The **v** flag must be set and the *MR* numbers are validated if the **v** flag has a value (the name of an *MR* number validation program). Diagnostics will occur if the **v** flag is not set or *MR* validation fails.

-h Causes `admin` to check the structure of the SCCS file, and to compare a newly computed check-sum (the sum of all the characters in the SCCS file except those in the first line) with the check-sum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

This option inhibits writing on the file, so that it nullifies the effect of any other options supplied. It is only meaningful when processing existing files.

-z The SCCS file check-sum is recomputed and stored in the first line of the SCCS file (see **-h**, above).

Note that use of this option on a truly corrupted file may prevent future detection of the corruption.

FILES

All SCCS file names must be of the form `s.`*file-name*. New SCCS files are given read-only permission mode (see `chmod`). Write permission in the pertinent directory is, of course, required to create a file. All writing done by `admin` is to a temporary x-file, called `x.`*file-name*, [see GET(SD_CMD)], created with read-only mode if the `admin` command is creating a new SCCS file, or with the same mode as the SCCS file if it exists. After successful execution of `admin`, the SCCS file is removed (if it exists), and the x-file is renamed with the name of the SCCS file. This ensures that changes are made to the SCCS file only if no errors occurred.

The command `admin` also makes use of a transient lock file (called `z.`*file-name*), which is used to prevent simultaneous updates to the SCCS file by different users. See GET(SD_CMD) for further information.

SEE ALSO

DELTA(SD_CMD), GET(SD_CMD), PRS(SD_CMD), WHAT(SD_CMD).

USAGE

General.

It is recommended that directories containing SCCS files be writeable by the owner only, and that SCCS files themselves be read-only. The mode of the directories allows only the owner to modify SCCS files contained in the directories. The mode of the SCCS files prevents any modification at all except by SCCS commands.

LEVEL

Level 1.

NAME

as — common assembler

SYNOPSIS

```
as [-o objfile] [-m] [-V] file-name
```

DESCRIPTION

The `as` command assembles the named file. The following options may be specified in any order:

`-o` *objfile*
: Put the output of the assembly in *objfile*. By default, the output file name is formed by removing the suffix, if there is one, from the input file name and appending a suffix.

`-m`
: Run the `m4` macro pre-processor on the input to the assembler.

`-V`
: Write the version number of the assembler being run on the standard error output.

SEE ALSO

CC(SD_CMD), LD(SD_CMD), M4(SD_CMD).

USAGE

General.

The command `cc` is the recommended interface to the assembler. The `as` command may not be present on all implementations of System V.

If the `-m` option (`m4` macro pre-processor invocation) is used, keywords for `m4` [see M4(SD_CMD)] cannot be used as symbols (variables, functions, labels) in the input file since `m4` cannot determine which are assembler symbols and which are real `m4` macros.

FUTURE DIRECTIONS

The `-Y` option is reserved for future use. It will be used to allow the user to specify the directories where the `m4` preprocessor, and the file of predefined macros are located.

Users will also be able to specify, by means of the `TMPDIR` environmental variable, the directory in which any temporary files are to be created.

These additions are part of the effort to eliminate hard-coded pathnames from the compilation system.

LEVEL

Level 1.

Optional.

NAME

cc — C compiler

SYNOPSIS

```
cc [ options ] file ...
```

DESCRIPTION

The `cc` command is the interface to the C compilation system. The system conceptually consists of a preprocessor, compiler, optimizer, assembler, and link-editor. The `cc` command processes the supplied options and then executes the various tools with the appropriate arguments.

The suffix of a file-name argument indicates how the file is to be treated. Files whose names end with `.c` are taken to be C source programs, and may be preprocessed, compiled, optimized, and link-edited. The compilation process may be stopped after the completion of any pass if the appropriate options are supplied. If the compilation process is allowed to complete the assembly phase, then an object program is produced; the object program for a source file called `xyz.c` is created in a file called `xyz.o`. However, the `.o` file is normally deleted if a single C program is compiled and loaded all at one go.

In the same way, arguments whose names end with `.s` are taken to be assembly source programs, and may be assembled and link-edited. Files with names ending in `.i` are taken to be preprocessed C source programs and may be compiled, optimized, assembled, and link-edited. Files whose names do not end in `.c`, `.s`, or `.i` are handed to the link-editor.

By default, if an executable file is produced (i.e., the link-edit phase is allowed to complete), the file is called `a.out`. This default name can be changed with the `-o` option (see below).

The following options are interpreted by `cc`:

`-c` Suppress the link edit phase of the compilation, and do not remove any object files that are produced.

`-f` Include floating-point support for systems without an automatically included floating-point implementation. This option is ignored on systems that do not need it.

`-g` Cause the compiler to generate additional information needed for the use of `sdb`.

`-o` *outfile*
Use the name `outfile`, instead of the default `a.out`, for the executable file produced. This is a link-editor option.

`-p` Arrange for the compiler to produce code that counts the number of times each routine is called; also, if link editing takes place, a profiled version of the standard C library is linked, and `monitor` [see MONITOR(SD_LIB)] is automatically called. A `mon.out` file will then be produce at normal termination of execution of the program. An

execution profile can then be generated by use of `prof`.

`-q` This option is reserved for specification of implementation specific profiling directives.

`-E` Run only `cpp` on the named C programs and send the result to the standard output.

`-F` This option is reserved for implementation specific optimization directives.

`-O` Do compilation phase optimization. This option will not affect files.

`-P` Run only `cpp` on the named C programs and leave the result on corresponding files suffixed `.i`. This option is passed to `cpp`.

`-S` Compile and do not assemble the named C programs, and leave the assembler-language output on corresponding files suffixed `.s`.

`-W`*c*,*arg1*[,*arg2* ...]
Hand off the argument[s] *argi* to phase *c* where *c* is one of `[p02al]` indicating preprocessor, compiler, optimizer, assembler, or link editor, respectively. For example, `-Wa,-m` passes `-m` to the assembler phase.

The `cc` command also recognizes the options `-C`, `-D`, `-I`, and `-U`, and passes them (and their associated arguments) directly to the preprocessor without using the `-W` option. Similarly, the loader options `-a`, `-l`, `-o`, `-r`, `-s`, `-u`, `-L`, and `-V` are recognized and passed directly to the loader. See CPP(SD_CMD) and LD(SD_CMD) for descriptions of these options.

Other arguments are taken to be C-compatible object programs, typically produced by an earlier `cc` or `pcc` run, or perhaps libraries of C-compatible routines, and are passed directly to the link-editor. These programs, together with the results of any compilations specified, are linked (in the order given) to produce an executable program with the name `a.out` (unless the `-o` link-editor option is used).

The standard C library is automatically available to the C program. Other libraries (including the math library) must be specified explicitly using the `-l` option with `cc`; see LD(SD_CMD) for details.

FILES

`file.c`	input file
`file.i`	preprocessed C source file
`file.o`	object file
`file.s`	assembly language file
`a.out`	link-edited (executable) output

SEE ALSO

CPP(SD_CMD), LD(SD_CMD), PROF(SD_CMD), SDB(SD_CMD), EXIT(BA_OS), MONITOR(SD_LIB).

USAGE

General.

Arbitrary length variable names are allowed in the C language, starting with System V Release 2.

Since the `cc` command usually creates files in the current directory during the compilation process, it is typically necessary to run the `cc` command in a directory in which a file can be created.

FUTURE DIRECTIONS

The `-Y` option is reserved for future use. It will be used to allow the user to specify the directories searched by the various components of `cc`.

Users will also be able to specify, by means of the `TMPDIR` environmental variable, the directory in which any temporary files are to be created.

These additions are part of the effort to eliminate hard-coded pathnames from the compilation system.

LEVEL

Level 1.

NAME

cflow – generate C flowgraph

SYNOPSIS

```
cflow [-r] [-ix] [-i_ ] [-dnum] files
```

DESCRIPTION

The command `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`. (The `-I`, `-D`, and `-U` options of the C-preprocessor are also understood by `cflow`.) Files suffixed with `.s` are assembled and information is extracted (as in `.o` files) from the symbol table. The output of all this processing 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 amount of indentation 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 file name and location counter under which the symbol appeared (e.g., `text`).

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.

The following options are interpreted by `cflow`:

`-r` Reverse the "caller:callee" relationship producing an inverted listing showing the callers of each function. The listing is also sorted in lexicographical order by callee.

`-ix` Include external and static data symbols. The default is to include only functions in the flowgraph.

`-i_` Include names that begin with an underscore. The default is to exclude these functions (and data if `-ix` is used).

`-d`*num* The *num* decimal integer indicates the depth at which the flowgraph is cut off. By default this is a very large number. Attempts to set the cutoff depth to a non-positive integer will be ignored.

SEE ALSO

CC(SD_CMD), LEX(SD_CMD), LINT(SD_CMD), YACC(SD_CMD).

USAGE

General.

Files produced by `lex` and `yacc` cause the reordering of line number declarations which can confuse `cflow`. To get proper results, feed `cflow` the `yacc` or `lex` input.

LEVEL

Level 1.

NAME

chroot — change root directory for a command

SYNOPSIS

```
/etc/chroot newroot command
```

DESCRIPTION

The command `chroot` executes the given `command`, **relative to the new root**. The meaning of any initial slashes (/) in path names is changed for a command and any of its children to `newroot`. Furthermore, the initial working directory is `newroot`.

This command is restricted to the super-user.

Notice that:

```
chroot newroot command >x
```

will create the file `x` relative to the original root, not the new one.

The new root path name is always relative to the current root: even if a `chroot` is currently in effect, the `newroot` argument is relative to the current root of the running process.

SEE ALSO

CHDIR(BA_OS)

USAGE

General.

The user should exercise caution when referencing special files in the new root file system.

LEVEL

Level 1.

NAME

cpp — the C language preprocessor

SYNOPSIS

```
LIBDIR/cpp [ option ... ] [ ifile [ ofile ] ]
```

DESCRIPTION

The command `cpp` is the C language preprocessor, which is invoked as the first pass of any C compilation using the `cc` command. Thus the output of `cpp` is designed to be in a form acceptable as input to the next pass of the C compiler.

`LIBDIR` is usually `/lib`.

The `cpp` command optionally accepts two file names as arguments; `ifile` and `ofile` are respectively the input and output for the preprocessor. They default to standard input and standard output if not supplied.

The following options to `cpp` are recognized:

`-P` Preprocess the input without producing the line control information used by the next pass of the C compiler.

`-C` By default, `cpp` strips C-style comments. If the `-C` option is specified, all comments (except those found on `cpp` directive lines) are passed along.

`-U`*name* Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.

`-D`*name*
`-D`*name*`=`*def*
Define *name* as if by a `#define` directive. If no `=def` is given, *name* is defined as 1. The `-D` option has lower precedence than the `-U` option. That is, if the same name is used in both a `-U` option and a `-D` option, the name will be undefined regardless of the order of the options.

`-I`*dir* Change the algorithm for searching for `#include` files whose names do not begin with `/` to look in `dir` before looking in the directories on the standard list. Thus, `#include` files whose names are enclosed in `""` will be searched for first in the directory of the file with the `#include` line, then in directories named in `-I` options, and last in directories on a standard list. For `#include` files whose names are enclosed in `<>`, the directory of the file with the `#include` line is not searched.

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 file name (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 `#`. Any number of blanks and tabs are allowed between the `#` and the directive. The directives are:

`#define` *name token-string*
Replace subsequent instances of *name* with *token-string.*

`#define` *name*`(` *arg*`, ..., ` *arg* `)` *token-string*
Notice that there can be no space between *name* and the `(`. Replace subsequent instances of *name* followed by a `(`, a list of comma-separated set of tokens, and a `)` by *token-string,* where each occurrence of an *arg* in the *token-string* is replaced by the corresponding set of tokens in the comma-separated list. When a macro with arguments is expanded, the arguments are placed into the expanded *token-string* unchanged. After the entire *token-string* has been expanded, `cpp` re-starts its scan for names to expand at the beginning of the newly created *token-string.*

`#undef` *name*
Cause the definition of *name* (if any) to be forgotten from now on. No additional tokens are permitted on the line after *name.*

`#include` *"filename"*
`#include` *<filename>*
Include 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. No additional tokens are permitted on the line after the final " or >.

`#line` *integer-constant "filename"*
Causes `cpp` to generate line control information for the next pass of the C compiler. *Integer-constant* is the line number of the next line and *filename* is the file where it comes from. If *"filename"* is not given, the current file name is unchanged. No additional tokens are permitted on the line after the final ".

`#endif`
Ends a section of lines begun by a test directive (`#if`, `#ifdef`, or `#ifndef`). Each test directive must have a matching `#endif`. No additional tokens are permitted on the line.

`#ifdef` *name*
The lines following will appear in the output if and only if *name* has been the subject of a previous `#define` without being the subject of an intervening `#undef`. No additional tokens are permitted on the line after *name.*

`#ifndef` *name*
The lines following will not appear in the output if and only if *name* has been the subject of a previous `#define` without being the subject of an intervening `#undef`. No additional tokens are permitted

on the line after *name*.

`#if` *constant-expression*
Lines following will appear in the output if and only if the *constant-expression* evaluates to non-zero. All binary non-assignment C operators, the `?:` operator, the unary `-`, `!`, and `@` operators are all 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`, which can be used in *constant-expression* in these two forms: `defined(`*name*`)` or `defined` *name*. This allows the utility of `#ifdef` and `#ifndef` in a `#if` directive. Only these operators, integer constants, and names which are known by `cpp` should be used in *constant-expression*. In particular, the `sizeof` operator is not available.

`#else`
The else part of an `#ifdef`, `#ifndef`, or `#if`. The lines preceding are ignored, and the lines following (upto the `#endif`) are included in the output if the test is false.

The test directives and the optional `#else` directives can be nested.

EXAMPLE

In order to test whether either of the two symbols *abc* and *def* are defined, use

```
#if defined (abc) || defined (def)
```

SEE ALSO

CC(SD_CMD).

USAGE

General.

The recommended way to invoke `cpp` is through the `cc` command. See M4(SD_CMD) for a general macro processor.

Include directives should avoid using hard-coded path-names: for example,
`#include <file.h>` should be used, rather than
`#include "/usr/include/file.h"`

FUTURE DIRECTIONS

The option `-Y` is reserved for future use. It will be used to specify a directory to be used instead of the standard list, when searching for `#include` files.

Users will also be able to specify, by means of the `TMPDIR` environmental variable, the directory in which any temporary files are to be created.

LEVEL

Level 1.

NAME

cxref — generate C program cross-reference

SYNOPSIS

`cxref [ options ] files`

DESCRIPTION

The command `cxref` analyzes a collection of C files and attempts to build a cross-reference table. Information from `#define` lines is included in the symbol table. A listing is produced on standard output of all symbols (auto, static, and global) in each file separately, or with the `-c` option, in combination. Each symbol contains an asterisk (*) before the declaring reference.

In addition to the `-D`, `-I`, and `-U` options (which are identical to their interpretation by `cc`) the following options are interpreted by `cxref`:

`-c` Print a combined cross-reference of all input files.

`-w`*num* Width option which formats output no wider than *num* (decimal) columns. This option will default to 80 if *num* is not specified or is less than 51.

`-o` *file* Direct output to named *file*.

`-s` Operate silently; does not print input file names.

SEE ALSO

CC(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

delta — make a delta (change) to an SCCS file

SYNOPSIS

```
delta [ -rSID ] [ -s ] [ -n ] [ -glist ] [
-m[mrlist] ] [ -y[comment] ] [ -p ] file ...
```

DESCRIPTION

The command `delta` is used to permanently introduce into the named SCCS file changes that were made to the file retrieved by `get` (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 path name does not begin with s.) and unreadable files are silently ignored. If a name of − is given, the standard input is read; in this case the −y option (see below) is required on the command line; if the −m option (see below) would normally be required, then it too is required on the command line. 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(SD_CMD)] that may be present in the SCCS file (see −m and −y keyletters below).

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 and will cause an error.

Keyletter arguments apply independently to each named file.

-r*SID* Uniquely identifies 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(SD_CMD)]. A diagnostic results if the specified SID is ambiguous, or, if necessary and 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.

-n Specifies retention of the edited *g-file* (normally removed at completion of delta processing).

-g*list* Specifies a *list* [see GET(SD_CMD) 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*] If the SCCS file has the **v** flag set [see ADMIN(SD_CMD)] then a Modification Request (MR) number *must* be supplied as the reason for creating the new delta.

If **-m** 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** keyletter).

MRs in a list are separated by blanks and/or tab characters. An unescaped new-line character terminates the MR list.

Note that if the **v** flag has a value, it is taken to be the name of a program which will validate the correctness of the MR numbers. If a non-zero exit status is returned from MR number validation program, `delta` terminates. (It is assumed that the MR numbers were not all valid.)

-y[*comment*] Arbitrary text used to describe the reason for making the delta. A null string is considered a valid *comment*.

If **-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 new-line character terminates the comment text.

-p Causes `delta` to print (on the standard output) the SCCS file differences before and after the delta is applied in `diff` format [see DIFF(BU_CMD)].

SEE ALSO

ADMIN(SD_CMD), GET(SD_CMD), PRS(SD_CMD), RMDEL(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

dis — disassembler

SYNOPSIS

```
dis [-o] [-V] [-L] [-F function ] [-l string ]
files
```

DESCRIPTION

The `dis` command produces an assembly language listing of each of its `file` arguments, each of which may be an object file or an archive of object files. The listing includes assembly statements and an octal or hexadecimal representation of the binary that produced those statements.

The following options are interpreted by the disassembler and may be specified in any order.

`-o` Will print numbers in octal. Default is hexadecimal.

`-V` Version number of the disassembler will be written to standard error.

`-L` Invokes a lookup of C source labels in the symbol table for subsequent printing.

`-F` *function* Disassembles only the named *function* in each object file specified on the command line. This option may be specified a number of times on the command line.

`-l` *string* Will disassemble the library file specified as *string*. For example, the command `dis -lm` will disassemble the math library.

SEE ALSO

AS(SD_CMD), CC(SD_CMD).

USAGE

General.

FUTURE DIRECTIONS

The `-s` option is reserved for future use. It will be used to specify symbolic disassembly.

LEVEL

Level 1.

Optional.

NAME

env — set environment for command execution

SYNOPSIS

```
env [-] [ name=value ] ...  [ command ]
```

DESCRIPTION

The command `env` obtains the current environment, modifies it according to its arguments, then executes the command with the modified environment. Arguments of the form `name=value` modify the execution environment: they are merged into the inherited environment before the command is executed. The `-` option causes the inherited environment to be ignored completely, so that the command is executed with exactly the environment specified by the arguments.

If no command is specified, the resulting environment is printed, one name-value pair per line.

SEE ALSO

SH(BU_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

get — get a version of an SCCS file

SYNOPSIS

```
get [ -r SID ] [ -c cutoff ] [ -e ] [ -b ] [ -i
list ] [ -x list ] [ -k ] [ -l [p] ] [ -p ] [ -s
] [ -m ] [ -n ] [ -g ] [ -t ] file ...
```

DESCRIPTION

The command *get* generates an ASCII text file from each named SCCS file according to the specifications given by its keyletter arguments, which begin with **-**. The arguments may be specified in any order, but all keyletter arguments apply to all named SCCS files. If a directory is named, `get` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of **-** is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed. Again, non-SCCS files and unreadable files are silently ignored.

The generated text is normally written into a file called the `g-file` whose name is derived from the SCCS file name by simply removing the leading **s.**; (see also **FILES**, below).

Each of the keyletter arguments is explained below as though only one SCCS file is to be processed, but the effects of any keyletter argument applies independently to each named file.

-r*SID* The SCCS Identification string (SID) of the version (delta) of an SCCS file to be retrieved. Table 1 below shows, for the most useful cases, what version of an SCCS file is retrieved (as well as the SID of the version to be eventually created by `delta` if the **-e** keyletter is also used), as a function of the SID specified.

-c*cutoff* *Cutoff* date-time, in the form:

YY[MM[DD[HH[MM[SS]]]]]

No changes (deltas) to the SCCS file which were created after the specified *cutoff* date-time are included in the generated ASCII text file. Units omitted from the date-time default to their maximum possible values; that is, **-c7502** is equivalent to **-c750228235959.** Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows one to specify a *cutoff* date in the form: "**-c77/2/2 9:22:25**".

-e Indicates that the `get` is for the purpose of editing or making a change (delta) to the SCCS file via a subsequent use of `delta`. The **-e** keyletter used in a `get` for a particular version (SID) of the SCCS file prevents further `gets` for editing

on the same SID until `delta` is executed or the **j** (joint edit) flag is set in the SCCS file. Concurrent use of `get` **−e** for different SIDs is always allowed.

If the `g-file` generated by `get` with an **−e** keyletter is accidentally ruined in the process of editing it, it may be regenerated by re-executing the `get` command with the **−k** keyletter in place of the **−e** keyletter.

SCCS file protection specified via the ceiling, floor, and authorized user list stored in the SCCS file are enforced when the **−e** keyletter is used.

−b Used with the **−e** keyletter to indicate that the new delta should have an SID in a new branch as shown in Table 1. This keyletter is ignored if the **b** flag is not present in the file or if the retrieved `delta` is not a leaf `delta`. (A leaf `delta` is one that has no successors on the SCCS file tree.)
Note: A branch `delta` may always be created from a non-leaf `delta`.

−i*list* A *list* of deltas to be included (forced to be applied) in the creation of the generated file. The *list* has the following syntax:

<list> ::= <range> | <list> , <range>
<range> ::= SID | SID − SID

SID, the SCCS Identification of a delta, may be in any form shown in the "SID Specified" column of Table 1. Partial SIDs are interpreted as shown in the "SID Retrieved" column of Table 1.

−x*list* A *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **−i** keyletter for the *list* format.

−k Suppresses replacement of identification keywords (see below) in the retrieved text by their value. The **−k** keyletter is implied by the **−e** keyletter.

−l[p] Causes a delta summary to be written into an `l-file`. If **−lp** is used then an `l-file` is not created; the delta summary is written on the standard output instead. See **FILES** for the format of the `l-file`.

−p Causes the text retrieved from the SCCS file to be written on the standard output. No `g-file` is created. All output which normally goes to the standard output goes to standard error instead, unless the **−s** keyletter is used, in which case it disappears.

-s Suppresses all output normally written on the standard output. However, fatal error messages (which always go to standard error) remain unaffected.

-m Causes each text line retrieved from the SCCS file to be preceded by the SID of the delta that inserted the text line in the SCCS file. The format is: SID, followed by a horizontal tab, followed by the text line.

-n Causes each generated text line to be preceded with the %M% identification keyword value (see below). The format is: %M% value, followed by a horizontal tab, followed by the text line. When both the **-m** and **-n** keyletters are used, the format is: %M% value, followed by a horizontal tab, followed by the **-m** keyletter generated format.

-g Suppresses the actual retrieval of text from the SCCS file. It is primarily used to generate an `l-file`, or to verify the existence of a particular SID.

-t Used to access the most recently created ("top") delta in a given release (e.g., **-r1**), or release and level (e.g., **-r1.2**).

For each file processed, `get` responds (on the standard output) with the SID being accessed and with the number of lines retrieved from the SCCS file.

If the **-e** keyletter is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated. If there is more than one named file or if a directory or standard input is named, each file name is printed (preceded by a new-line) before it is processed. If the **-i** keyletter is used included deltas are listed following the notation "Included"; if the **-x** keyletter is used, excluded deltas are listed following the notation "Excluded".

TABLE 1. Determination of SCCS Identification String

SID* Specified	**-b** Keyletter Used†	Other Conditions	SID Retrieved	SID of Delta to be Created
none‡	no	R defaults to mR	mR.mL	mR.(mL+1)
none‡	yes	R defaults to mR	mR.mL	mR.mL.(mB+1).1
R	no	R > mR	mR.mL	R.1***
R	no	R = mR	mR.mL	mR.(mL+1)
R	yes	R > mR	mR.mL	mR.mL.(mB+1).1
R	yes	R = mR	mR.mL	mR.mL.(mB+1).1
R	–	R < mR and R does *not* exist	hR.mL**	hR.mL.(mB+1).1
R	–	Trunk succ.# in release > R	R.mL	R.mL.(mB+1).1

		and R exists		
R.L	no	No trunk succ.	R.L	R.(L+1)
R.L	yes	No trunk succ.	R.L	R.L.(mB+1).1
R.L	–	Trunk succ. in release ⩾ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch succ.	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch succ.	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch succ.	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch succ.	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch succ.	R.L.B.S	R.L.(mB+1).1

* "R", "L", "B", and "S" are the "release", "level", "branch", and "sequence" components of the SID, respectively; "m" means "maximum". Thus, for example, "R.mL" means "the maximum level number within release R"; "R.L.(mB+1).1" means "the first sequence number on the *new* branch (i.e., maximum branch number plus one) of level L within release R". Note that if the SID specified is of the form "R.L", "R.L.B", or "R.L.B.S", each of the specified components *must* exist.

** "hR" is the highest *existing* release that is lower than the specified, *nonexistent*, release R.

*** This is used to force creation of the *first* delta in a *new* release.

\# Successor.

† The **−b** keyletter is effective only if the **b** flag is present in the file. An entry of − means "irrelevant".

‡ This case applies if the **d** (default SID) flag is *not* present in the file. If the **d** flag *is* present in the file, then the SID obtained from the **d** flag is interpreted as if it had been specified on the command line. Thus, one of the other cases in this table applies.

IDENTIFICATION KEYWORDS

Identifying information is inserted into the text retrieved from the SCCS file by replacing *identification keywords* with their value wherever they occur. The following keywords may be used in the text stored in an SCCS file:

Keyword	*Value*
%M%	Module name: either the value of the **m** flag in the file, or if absent, the name of the SCCS file with the leading **s.** removed.
%I%	SCCS identification (SID) (%R%.%L%.%B%.%S%) of the retrieved text.
%R%	Release.
%L%	Level.
%B%	Branch.
%S%	Sequence.
%D%	Current date (YY/MM/DD).
%H%	Current date (MM/DD/YY).
%T%	Current time (HH:MM:SS).
%E%	Date newest applied delta was created (YY/MM/DD).

%G%	Date newest applied delta was created (MM/DD/YY).
%U%	Time newest applied delta was created (HH:MM:SS).
%Y%	Module type: value of the **t** flag in the SCCS file.
%F%	SCCS file name.
%P%	Fully qualified SCCS file name.
%Q%	The value of the **q** flag in the file.
%C%	Current line number. This keyword is intended for identifying messages output by the program such as "this should not have happened" type errors. It is *not* intended to be used on every line to provide sequence numbers.
%Z%	The 4-character string @(#) recognizable by `what`.
%W%	A shorthand notation for constructing `what` strings. %W% = %Z%%M%<horizontal-tab>%I%
%A%	Another shorthand notation for constructing *what*(SD_CMD) strings. %A% = %Z%%Y% %M% %I%%Z%

FILES

Several auxiliary files may be created by `get`. These files are known generically as the `g-file`, `l-file`, `p-file`, and `z-file`. The letter before the hyphen is called the tag. An auxiliary file name is formed from the SCCS file name: the last component of all SCCS file names must be of the form **s.***module-name*, the auxiliary files are named by replacing the leading **s** with the tag. The `g-file` is an exception to this scheme: the `g-file` is named by removing the **s.** prefix. For example, `s.xyz.c`, the auxiliary file names would be `xyz.c`, `l.xyz.c`, `p.xyz.c`, and `z.xyz.c`, respectively.

The `g-file`, which contains the generated text, is created in the current directory (unless the **−p** keyletter is used). A `g-file` is created in all cases, whether or not any lines of text were generated by the `get`. It is owned by the real user. If the **−k** keyletter is used or implied it is writeable by the owner only (read-only for everyone else); otherwise it is read-only. Only the real user need have write permission in the current directory.

The `l-file` contains a table showing which deltas were applied in generating the retrieved text. The `l-file` is created in the current directory if the **−l** keyletter is used; it is read-only and it is owned by the real user. Only the real user need have write permission in the current directory.

Lines in the `l-file` have the following format:

a. A blank character if the delta was applied;
   * otherwise.
b. A blank character if the delta was applied or was not applied and ignored;
   * if the delta was not applied and was not ignored.
c. A code indicating a "special" reason why the delta was or was not applied:
   "I": Included.

"X": Excluded.
"C": Cut off (by a **-c** keyletter).

d. Blank.
e. SCCS identification (SID).
f. Tab character.
g. Date and time (in the form YY/MM/DD HH:MM:SS) of creation.
h. Blank.
i. Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one horizontal tab character. A blank line terminates each entry.

The `p-file` is used to pass information resulting from a `get` with an **-e** keyletter along to `delta`. Its contents are also used to prevent a subsequent execution of `get` with an **-e** keyletter for the same SID until `delta` is executed or the joint edit flag, **j**, is set in the SCCS file. The `p-file` is created in the directory containing the SCCS file and the effective user must have write permission in that directory. It is writeable by owner only, and it is owned by the effective user. The format of the `p-file` is: the gotten SID, followed by a blank, followed by the SID that the new delta will have when it is made, followed by a blank, followed by the login name of the real user, followed by a blank, followed by the date-time the `get` was executed, followed by a blank and the **-i** keyletter argument if it was present, followed by a blank and the **-x** keyletter argument if it was present, followed by a new-line. There can be an arbitrary number of lines in the `p-file` at any time; no two lines can have the same new delta SID.

The `z-file` serves as a *lock-out* mechanism against simultaneous updates. Its contents are the binary process ID of the command (i.e., `get`) that created it. The `z-file` is created in the directory containing the SCCS file for the duration of `get`. The same protection restrictions as those for the `p-file` apply for the `z-file`. The `z-file` is created read-only.

SEE ALSO

ADMIN(SD_CMD), DELTA(SD_CMD), PRS(SD_CMD), WHAT(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

ld — link editor for object files

SYNOPSIS

```
ld [ options ] file ...
```

DESCRIPTION

The `ld` command combines several object files into one, performs relocation, resolves external symbols, and supports symbol table information for symbolic debugging. In the simplest case, the names of several object programs are given, and `ld` combines them, producing an object module that can either be executed or, if the `-r` option is specified, used as input for a subsequent `ld` run. The output of `ld` is left in `a.out`. By default this file is executable if no errors occurred during the load. If any input file `filename`, is not an object file, `ld` assumes it is an archive library.

If any argument is a library, it is searched at the point it is encountered in the argument list. Only those routines defining an unresolved external reference are loaded. The library (archive) symbol table is searched to resolve external references which can be satisfied by library members. The ordering of library members is unimportant, unless there exist multiple library members defining the same external symbol.

The following options are recognized by `ld`:

`-e` *epsym* Set the default entry point address for the output file to be that of the symbol *epsym*.

`-l`*x* Search the library which has the abbreviation *x* (e.g., *-lm* to search the math library). A library is searched when its name is encountered, so the placement of a `-l` option is significant.

`-o` *outfile*
Produce an output object file by the name *outfile*. The name of the default object file is `a.out`.

`-r` Retain relocation entries in the output object file. Relocation entries must be saved if the output file is to become an input file in a subsequent `ld` run. The link editor will not complain about unresolved references, and the output file will not be made executable.

`-s` Strip all symbolic information from the output object file.

`-u` *symname*
Enter *symname* as an undefined symbol in the symbol table. This is useful for loading entirely from a library, since initially the symbol table is empty and an unresolved reference is needed to force the loading of the first routine.

`-L` *dir*
Change the algorithm of searching for the library *x* to look in *dir* before looking in the default library directories. This option is effective only if it precedes the `-l` option on the command line.

`-V` Output a message giving information about the version of `ld` being used.

FILES

`"a.out"`	output file

SEE ALSO

AR(BU_CMD), CC(SD_CMD), STRIP(SD_CMD).

USAGE

General.

When the link editor is called through `cc`, a startup routine is linked with the user's program. This routine calls `exit( )` after execution of the main program. If the user calls the link editor directly, then the user must ensure that the program always calls `exit( )` rather than falling through the end of the entry routine.

The symbols `etext`, `edata`, and `end` are reserved and are defined by the link editor. It is erroneous for a user program to redefine them.

FUTURE DIRECTIONS

The option `-Y` is reserved for future use. It will be used to specify a directory to be used instead of the standard list, when searching for libraries.

Users will also be able to specify, by means of the `TMPDIR` environmental variable, the directory in which any temporary files are to be created.

LEVEL

Level 1.

NAME

lex — generate programs for simple lexical analysis of text

SYNOPSIS

```
lex [ -ctvn ] [ file ] ...
```

DESCRIPTION

The command `lex` generates programs to be used in lexical processing of character input and may be used as an interface to `yacc`.

The input `file`(s), which contain `lex` source code, contain a table of regular expressions each with a corresponding action in the form of a C program fragment. Multiple input `files` are treated as a single file. When `lex` processes `file`(s), this source is translated into a C program. Normally `lex` writes the program it generates to the file `lex.yy.c`. If the `-t` option is used, the resulting program is written instead to the standard output. When the program generated by `lex` is compiled and executed, it will read character input from the standard input and partition it into strings that match the given expressions. When an expression is matched, the input string was matched is left in an external character array `yylex` and the expressions corresponding program fragment, or action, is executed. During pattern matching the set of patterns will be searched for a match in the order in which they appeared in the `lex` source and the single longest possible matchwill be chosen at any point in time. Among rules that match the same number of characters, the rule given first will be matched.

The program generated by `lex`, e.g., `lex.yy.c`, should be compiled and loaded with the `lex` library (using the `-ll` option with `cc`.

The option `-c` indicates C language actions and is the default, `-t` causes the program generated to be written instead to standard output, `-v` provides a one-line summary of statistics of the finite state machine generated, `-n` will not print out the `-` summary.

Certain table sizes for the resulting finite state machine can be set in the definitions section:

%p *n*	number of positions is *n*
%n *n*	number of states is *n*
%e *n*	number of parse tree nodes is *n*
%a *n*	number of transitions is *n*
%k *n*	number of packed character classes is *n*
%o *n*	size of the output array is *n*

The use of one or more of the above automatically implies the `-v` option, unless the `-n` option is used.

The general format of `lex` source is:

{definitions}
%%
{rules}
%%

{user subroutines}

The definitions and the user subroutines may be omitted. The first %% is required to mark the beginning of the rules (regular expressions and actions); the second %% is required only if user subroutines follow.

Any line in the source beginning with a blank is assumed to contain only C text and is copied to `lex.yy.c`; if it precedes %% it is copied into the external definition area of the `lex.yy.c` file. Anything included between lines containing only %{ **and** %} is copied unchanged to `lex.yy.c` and the delimiter lines are discarded. Anything after the third %% delimiter is copied to `lex.yy.c`.

Definitions

Definitions must appear before the first %% delimiter. Any line in this section not contained between %{ **and** %} lines and beginning in column 1 is assumed to define a `lex` substitution string. The format of these lines is

name substitute

The *name* must begin with a letter and be followed by at least one blank or tab. The substitute will replace the string {*name*} when it is used in a rule. The curly braces do not imply parentheses; only string substitution is done.

Rules

The rules in `lex` source files are a table in which the left column contains regular expressions and the right column contains actions and program fragments to be executed when the expressions are recognized.

```
re whitespace action
re whitespace action
    ...
```

Because the regular expression, (*re*), portion of a rule is terminated by the first blank or tab, any blank or tab used within a regular expression must be quoted (its special meaning escaped). That is, it must appear within double quotes, square brackets or must be preceded by a backslash character.

The program fragment which is the action associated with a particular *re* may extend across several lines if it is enclosed in curly braces:

```
re whitespace { program statement
                program statement }
```

Regular Expressions

The `lex` command supports the sets of regular expressions recognized by `ed` and `awk`, and some additional expressions. Some characters have special meanings when used in an *re* and are called regular expression operators. Below is a table of expressions supported by `lex`.

Regular Expression	Pattern Matched
c	the character *c* where *c* is not a special character.
\c	the character *c* where *c* is any character.
"c"	the character *c* where *c* is any character except \.
^	the beginning of the line being compared.
$	the end of the line being compared.
.	any character in the input but newline
[s]	any character in the set *s* where *s* is a sequence of characters and/or a range of characters, *c-c*.
[^s]	any character not in the set *s*, where *s* is defined as above.
r*	zero or more successive occurrences of the regular expression *r*.
r+	one or more successive occurrences of the regular expression *r*.
r?	zero or one occurrence of the regular expression *r*.
(r)	the regular expression *r*. (Grouping)
rx	the occurrence of regular expression *r* followed by the occurrence of regular expression *x*. (Concatenation)
r|x	the occurrence of regular expression *r* or the occurrence of regular expression *x*.
<s>r	the occurrence of regular expression *r* only when the program is in start condition (state) *s*.
r/x	the occurrence of regular expression *r* only if it is followed by the occurence of regular expression *x*. (Note this is *r* in the context of *x* and only *r* is matched.)
{S}	the substitution of *S* from the *Definitions* section.
r{m,n}	*m* through *n* successive occurrences of the regular expression *r*.

The notation *r{m,n}* in a rule indicates between *m* and *n* instances of regular expression *r*. It has higher precedence than |, but lower than *, ?, +, and concatenation.

The character ^ at the beginning of an expression permits a successful match only immediately after a new-line, and the character $ at the end of an expression requires a trailing new-line.

The character / in an expression indicates trailing context; only the part of the expression up to the slash is returned in **yytext**, but the remainder of the expression must follow in the input stream. An operator character may be used as an ordinary symbol if it is within double quotes, "c"; preceded by \, \c; or is within square brackets, [c]. Two operators have special meaning when used within square brackets. A - denotes a range, [c-c], unless it is just after the open bracket or before the closing bracket, [-c] or [c-] in which case it has no special meaning. When used within brackets, ^ has the meaning "complement of" if it immediately follows the open bracket, [^c], elsewhere between brackets, [c^], it stands for the ordinary character ^. The special meaning of the \ operator can be escaped *only* by preceding it with another \.

Actions

The default action when a string in the input to a **lex.yy.c** program is *not matched* by any expression is to copy the string to the output. Because the default behavior of a program generated by `lex` is to read the input and copy it to the output, a minimal `lex` source program that has just %% will generate a **C** program that simply copies the input to the output unchanged. A null **C** statement, the statement ';', may be specified as an action in a rule. Any string in the *lex.yy.c* input that matches the pattern portion of such a rule, will be effectively ignored or skipped.

Three special actions are available, **|**, **REJECT**, and **ECHO**. The action **|** means that the action for the next rule is the action for this rule. **ECHO** prints the string *yytext* on the output. Normally only a single expression is matched by a given string in the input. REJECT means "continue to the next expression that matches the current input" and causes whatever rule was second choice after the current rule to be executed for the same input. Thus, it allows multiple rules to be matched and executed for one input string or overlapping input strings. For example, given the expressions xyz and yz and the input xyz, normally only one pattern, xyz would match and the next attempted match would start at z. If the last action in the xyz rule is REJECT, both this rule and the yz rule would be executed.

The `lex` command provides several routines that can be used in the lex source program: **yymore()**, **yyless(n)**, **input()**, **output(c)**, and **unput(c)**.

The function **yymore()** may be called to indicate that the next input string recognized is to be concatenated onto the end of the current string in **yytext** rather than overwriting it in **yytext**.

yyless(n) returns to the input some of the characters matched by the currently successful expression. The argument "n" indicates the number of initial characters in **yytext** to be retained; the remaining trailing characters in **yytext** are returned to the input.

input() returns the next character from the input. **input** returns 0 on end of file.

unput(c) pushes the character c back onto the input stream to be read later by **input()**.

output(c) writes the character c on the output.

To perform custom processing when the end of input is reached, a user may supply their own **yywrap()** function. **yywrap()** is called whenever **lex.yy.c** reaches an end-of-file. If **yywrap()** returns a one, **lex.yy.c** continues with the normal wrap-up on end of input. The default **yywrap()** always returns a one. If the user wants **lex.yy.c** to continue processing with another source of input, then a **yywrap()** must be supplied that arranges for the new input and returns a zero. These routines may be redefined by the user.

The external names generated by `lex` all begin with the prefix **yy** or **YY**.

The program generated by *lex* is named **yylex()**; if the user does not supply a main routine, the default **main()** routine calls **yylex()**. If the user supplies a **main()** routine, it should call **yylex()**.

EXAMPLE

```
D       [0-9]
%%
if      printf("IF statement\n");
[a-z]+  printf("tag, value %s\n",yytext);
0{D}+   printf("octal number %s\n",yytext);
{D}+    printf("decimal number %s\n",yytext);
"++"    printf("unary op\n");
"+"     printf("binary op\n");
"/*"    {    loop:
             while (input() != '*');
             switch (input())
                  {
                  case '/': break;
                  case '*': unput('*');
                  default: go to loop;
                  }
             }
```

FILES

lex.yy.c.

SEE ALSO

CC(SD_CMD), YACC(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

lint — a C program checker

SYNOPSIS

```
lint [ options ] file ...
```

DESCRIPTION

The command `lint` attempts to detect features of the C program files that are likely to be bugs, non-portable, or wasteful. It also checks type usage more strictly than the compilers. Among the things that are currently detected are unreachable statements, loops not entered at the top, automatic variables declared and not used, and logical expressions whose value is constant. Moreover, the usage of functions is checked to find functions that return values in some places and not in others, functions called with varying numbers or types of arguments, and functions whose values are not used or whose values are used but none returned.

The options are described below. Note, however, that the options `-c` and `-o` are new to System V Release 2.

Arguments whose names end with `.c` are taken to be C source files. The following behavior is new in System V Release 2.

Arguments whose names end with `.ln` are taken to be the result of an earlier invocation of `lint` with either the `-c` or the `-o` option used. The `.ln` files are analogous to `.o` (object) files that are produced by the `cc` command when given a `.c` file as input.

Files with other suffixes are warned about and ignored.

The command `lint` will take all the `.c`, `.ln`, files, and `llib-l`*x*(specififed by `-l`*x*), and process them in their command line order. By default, `lint` appends the standard C lint library to the end of the list of files. However, if the `-p` option is used, the portable C lint library (`llib-port.ln`) is appended instead. When the `-c` option is not used, the second pass of `lint` checks this list of files for mutual compatibility. When the `-c` option is used, the `.ln` files and the lint libraries are ignored.

Any number of `lint` options may be used, in any order, intermixed with file-name arguments. The following options are used to suppress certain kinds of complaints:

`-a` Suppress complaints about assignments of long values to variables that are not long.

`-b` Suppress complaints about `break` statements that cannot be reached. (Programs produced by `lex` or `yacc` will often result in many such complaints).

`-h` Do not apply heuristic tests that attempt to intuit bugs, improve style, and reduce waste.

`-u` Suppress complaints about functions and external variables used and not defined, or defined and not used. (This option is suitable for running `lint` on a subset of files of a larger program).

`-v` Suppress complaints about unused arguments in functions.

`-x` Do not report variables referred to by external declarations but never used.

The following arguments alter `lint`'s behavior:

`-l`*x* Include additional lint library `x` (e.g., `-lm` for the math library).

`-n` Do not check compatibility against either the standard or the portable lint library.

`-p` Attempt to check portability.

`-c` Cause `lint` to produce a `.ln` file for every `.c` file on the command line. These `.ln` files are the product of `lint`'s first pass only, and are not checked for inter-function compatibility.

`-o` *lib* Cause `lint` to create a lint library with the name `lib`. The `-c` option nullifies any use of the `-o` option. The lint library produced is the input that is given to `lint`'s second pass. The `-o` option simply causes this file to be saved in the named lint library. To produce the lint library without extraneous messages, use of the `-x` option is suggested. The `-v` option is useful if the source file(s) for the lint library are just external interfaces. These option settings are also available through the use of "lint comments" (see below).

The `-D`, `-U`, and `-I` options of `cpp` [see CPP(SD_CMD)] are also recognized as separate arguments.

(The following is new to System V Release 2.) The `-g` and `-O` options of `cc` are also recognized as separate arguments. These options are ignored, but, by recognizing these options, `lint`'s behavior is closer to that of the `cc` command. Other options are warned about and ignored. The preprocessor symbol "lint" is defined to allow certain questionable code to be altered or removed for `lint`. Therefore, the symbol "lint" should be thought of as a reserved word for all code that is planned to be checked by `lint`.

Certain conventional comments in the C source will change the behavior of `lint`:

/*NOTREACHED*/
: at appropriate points stops comments about unreachabie code. (This comment is typically placed just after calls to functions like `exit`.

/*VARARGS*n*/
: suppresses the usual checking for variable numbers of arguments in the following function declaration. The data types of the first

n arguments are checked; a missing *n* is taken to be 0.

/*ARGSUSED*/
turns on the `-v` option for the next function.

/*LINTLIBRARY*/
at the beginning of a file shuts off complaints about unused functions and function arguments in this file. This is equivalent to using the `-v` and `-x` options.

The command `lint` produces its first output on a per-source-file basis. Complaints regarding included files are collected and printed after all source files have been processed. Finally, if the `-c` option is not used, information gathered from all input files is collected and checked for consistency. At this point, if it is not clear whether a complaint stems from a given source file or from one of its included files, the source file name will be printed followed by a question mark.

The behavior of the `-c` and the `-o` options allows for incremental use of `lint` on a set of C source files. Generally, `lint` is invoked once for each source file with the `-c` option. Each of these invocations produces a `.ln` file which corresponds to the `.c` file, and prints all messages that are about just that source file. After all the source files have been separately run through `lint`, it is invoked once more (without the `-c` option), listing all the `.ln` files with the needed `-lx` options. This will print all the inter-file inconsistencies. This scheme works well with `make`; it allows `make` to be used to `lint` only the source files that have been modified since the last time the set of source files were checked by `lint`.

SEE ALSO

CC(SD_CMD), CPP(SD_CMD), MAKE(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

lorder — find ordering relation for an object library

SYNOPSIS

```
lorder file ...
```

DESCRIPTION

The input is one or more object or library archive *files* [see AR(BU_CMD)]. The standard output is a list of pairs of object file names, meaning that the first file of the pair refers to external identifiers defined in the second. The output may be processed by `tsort` to find an ordering of a library suitable for one-pass access by the link editor `ld` Note that `ld` is capable of multiple passes over an archive in the portable archive format and does not require that `lorder` be used when building an archive. The usage of the `lorder` command may, however, allow for a slightly more efficient access of the archive during the link edit process.

EXAMPLE

The following example builds a new library from existing `.o` files.

```
ar -cr library `lorder *.o | tsort`
```

SEE ALSO

AR(BU_CMD), LD(SD_CMD), TSORT(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

m4 — macro processor

SYNOPSIS

```
m4 [ options ] [ file ... ]
```

DESCRIPTION

The command `m4` is a macro processor intended as a front end for Ratfor, C, and other languages. Each of the argument files is processed in order; if there are no files, or if a file name is `-`, the standard input is read. The processed text is written on the standard output.

The options and their effects are as follows:

`-s` Enable line sync output for the C preprocessor (i.e., `#line` directives).

This option must appear before any file names and before the following options.

`-D`*name*`[=`*val*`]`
Defines *name* to *val* or to null in *val*'s absence.

`-U`*name* undefines *name*.

Macro calls have the form:

```
name(arg1,arg2, ..., argn)
```

The `(` must immediately follow the name of the macro. If the name of a defined macro is not followed by a `(`, it is deemed to be a call of that macro with no arguments. Potential macro names consist of alphabetic letters, digits, and underscore `_`, where the first character is not a digit.

Leading unquoted blanks, tabs, and new-lines are ignored while collecting arguments. Left and right single quotes are used to quote strings. The value of a quoted string is the string stripped of the quotes.

When a macro name is recognized, its arguments are collected by searching for a matching right parenthesis. If fewer arguments are supplied than are in the macro definition, the trailing arguments are taken to be null. Macro evaluation proceeds normally during the collection of the arguments, and any commas or right parentheses which happen to turn up within the value of a nested call are as effective as those in the original input text. After argument collection, the value of the macro is pushed back onto the input stream and rescanned.

The command `m4` makes available the following built-in macros. They may be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

`define` the second argument is installed as the value of the macro whose name is the first argument. Each occurrence of `$`*n* in the replacement text, where *n* is a digit, is replaced by the *n*-th

argument. Argument 0 is the name of the macro; missing arguments are replaced by the null string; `$#` is replaced by the number of arguments; `$*` is replaced by a list of all the arguments separated by commas; `$@` is like `$*`, but each argument is quoted (with the current quotes).

`undefine` removes the definition of the macro named in its argument.

`defn` returns the quoted definition of its argument(s). It is useful for renaming macros, especially built-ins.

`pushdef` like *define,* but saves any previous definition.

`popdef` removes current definition of its argument(s), exposing the previous one, if any.

`ifdef` if the first argument is defined, the value is the second argument, otherwise the third. If there is no third argument, the value is null.

`shift` returns all but its first argument. The other arguments are quoted and pushed back with commas in between. The quoting nullifies the effect of the extra scan that will subsequently be performed.

`changequote` change quote symbols to the first and second arguments. The symbols may be up to five characters long. the command `changequote` without arguments restores the original values (i.e., `‘’`).

`changecom` change left and right comment markers from the default # and new-line. With no arguments, the comment mechanism is effectively disabled. With one argument, the left marker becomes the argument and the right marker becomes new-line. With two arguments, both markers are affected. Comment markers may be up to five characters long.

`divert` The command `m4` maintains 10 output streams, numbered 0-9. The final output is the concatenation of the streams in numerical order; initially stream 0 is the current stream. The `divert` macro changes the current output stream to its (digit-string) argument. Output diverted to a stream other than 0 through 9 is discarded.

`undivert` causes immediate output of text from diversions named as arguments, or all diversions if no argument. Text may be undiverted into another diversion. Undiverting discards the diverted text.

`divnum` returns the value of the current output stream.

`dnl` reads and discards characters up to and including the next new-line.

`ifelse` has three or more arguments. If the first argument is the same string as the second, then the value is the third argument. If not, and if there are more than four arguments, the process is repeated with arguments 4, 5, 6 and 7. Otherwise, the value is either the fourth string, or, if it is not present, null.

`incr` returns the value of its argument incremented by 1. The value of the argument is calculated by interpreting an initial digit-string as a decimal number.

`decr` returns the value of its argument decremented by 1.

`eval` evaluates its argument as an arithmetic expression, using 32-bit arithmetic. Operators include `+`, `-`, `*`, `/`, `%`, `^` (exponentiation), bitwise `&`, `|`, `^`, and `;` relationals; parentheses. Octal and hex numbers may be specified as in C. The second argument specifies the radix for the result; the default is 10. The third argument may be used to specify the minimum number of digits in the result.

`len` returns the number of characters in its argument.

`index` returns the position in its first argument where the second argument begins (zero origin), or −1 if the second argument does not occur.

`substr` returns a substring of its first argument. The second argument is a zero origin number selecting the first character; the third argument indicates the length of the substring. A missing third argument is taken to be large enough to extend to the end of the first string.

`translit` transliterates the characters in its first argument from the set given by the second argument to the set given by the third. No abbreviations are permitted.

`include` returns the contents of the file named in the argument.

`sinclude` is identical to `include`, except that it says nothing if the file is inaccessible.

`syscmd` executes the system command given in the first argument. No value is returned.

`sysval` is the return code from the last call to `syscmd`.

maketemp fills in a string of XXXXX in its argument with the current process ID.

m4exit causes immediate exit from `m4`. Argument 1, if given, is the exit code; the default is 0.

m4wrap argument 1 will be pushed back at final EOF; example: m4wrap(`cleanup()')

errprint — prints its argument on the diagnostic output file.

dumpdef — prints current names and definitions, for the named items, or for all if no arguments are given.

traceon — with no arguments, turns on tracing for all macros (including built-ins). Otherwise, turns on tracing for named macros.

traceoff — turns off trace globally and for any macros specified. Macros specifically traced by `traceon` can be untraced only by specific calls to `traceoff`.

SEE ALSO

CC(SD_CMD), CPP(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

make — maintain, update, and regenerate groups of programs

SYNOPSIS

```
make [-f makefile] [-p] [-i] [-k] [-s] [-r] [-n]
[-e] [-t] [-q] [ name ... ]
```

DESCRIPTION

The options are interpreted as follows:

`-f` Description file name. The argument `makefile` is assumed to be the name of a description file. A file name of — denotes the standard input.

`-p` Print out the complete set of macro definitions and target descriptions.

`-i` Ignore error codes returned by invoked commands. This mode is entered if the fake target name **.IGNORE** appears in the description file.

`-k` Abandon work on the current entry if it fails, but continue on other branches that do not depend on that entry.

`-s` Silent mode. Do not print command lines before executing. This mode is also entered if the fake target name **.SILENT** appears in the description file.

`-r` Do not use the built-in rules.

`-n` No execute mode. Print commands, but do not execute them. Even lines beginning with an `@` are printed.

`-e` Environmental variables override assignments within makefiles.

`-t` Touch the target files (causing them to be up-to-date) rather than issue the usual commands.

`-q` Question. The `make` command returns a zero or non-zero status code depending on whether the target file is or is not up-to-date.

The following target names may be defined in the `makefile`, and are interpreted as follows:

.DEFAULT If a file must be made but there are no explicit commands or relevant built-in rules, the commands associated with the name **.DEFAULT** are used if it exists.

.PRECIOUS Dependents of this target will not be removed when quit or interrupt are hit.

.SILENT Same effect as the `-s` option.

.IGNORE Same effect as the `-i` option.

The command `m` executes commands in `makefile` to update one or more target names. The argument `name` is typically a program. If no `-f` option is present, `makefile`, `Makefile`, and the SCCS files `s.makefile`, and `s.Makefile` are tried in order. If `makefile` is `-`, the standard input is used. More than one `-fmakefile` argument pair may appear.

The command `make` updates a target only if its dependents are newer than the target. All prerequisite files of a target are added recursively to the list of targets. Missing files are deemed to be out-of-date.

The argument `makefile` contains a sequence of entries that specify dependencies. The first line of an entry is a blank-separated, non-null list of targets, then a `:`, then a (possibly null) list of prerequisite files or dependencies. Text following a `;` and all following lines that begin with a tab are commands to be executed to update the target. The first line that does not begin with a tab or `#` begins a new dependency or macro definition. Commands may be continued across lines with the <backslash><new-line> sequence. Everything printed by make (except the initial tab) is passed directly to the command interpreter as is.

The symbols `#` and new-line surround comments.

The following `makefile` says that `pgm` depends on two files `a.o` and `b.o`, and that they in turn depend on their corresponding source files (`a.c` and `b.c`) and a common file `incl.h`:

```
pgm:  a.o  b.o  cc  a.o  b.o  -o  pgm
a.o:  incl.h  a.c  cc  -c  a.c
b.o:  incl.h  b.c  cc  -c  b.c
```

Command lines are executed one at a time. The first one or two characters in a command can be the following: `-`, `@`, `-@`, or `@-`. If `@` is present, printing of the command is suppressed. If `-` is present, `make` ignores an error. A line is printed when it is executed unless the `-s` option is present, or the entry **.SILENT:** is in `makefile`, or unless the initial character sequence contains a `@`. The `-n` option specifies printing without execution; however, if the command line has the string `$(MAKE)` in it, the line is always executed (see discussion of the `MAKEFLAGS` macro under **Environment**). The `-t` (touch) option updates the modified date of a file without executing any commands.

Commands returning non-zero status normally terminate `make`. If the `-i` option is present, or the entry **.IGNORE:** appears in `makefile`, or the initial character sequence of the command contains `-`, the error is ignored. If the `-k` option is present, work is abandoned on the current entry, but continues on other branches that do not depend on that entry.

Interrupt and quit cause the target to be deleted unless the target is a dependent of the special name **.PRECIOUS**.

Environment

The environment is read by `make`. All variables are assumed to be macro definitions and processed as such. The environmental variables are processed before any makefile and after the internal rules; thus, macro assignments in a makefile override environmental variables. The `-e` option causes the environment to override the macro assignments in a makefile.

The environmental variable `MAKEFLAGS` is processed by `make` as containing any legal input option (except `-f` and `-p`) defined for the command line. Further, upon invocation, `make` "invents" the variable if it is not in the environment, puts the current options into it, and passes it on to invocations of commands. Thus, `MAKEFLAGS` always contains the current input options. This proves very useful for "super-makes". In fact, as noted above, when the `-n` option is used, the command `$(MAKE)` is executed anyway; hence, one can perform a `make -n` recursively on a whole software system to see what would have been executed. This is because the `-n` is put in `MAKEFLAGS` and passed to further invocations of `$(MAKE)`. This is one way of debugging all of the makefiles for a software project without actually doing anything.

Macros

Entries of the form `string1 = string2` are macro definitions. The macro `string2` is defined as all characters up to a comment character or an unescaped new-line. Subsequent appearances of `$(string1[:`*subst1*`=[`*subst2*`]])` are replaced by `string2`. The parentheses are optional if a single character macro name is used and there is no substitute sequence. The optional `:subst1=`*subst2* is a substitute sequence. If it is specified, all non-overlapping occurrences of `subst1` in the named macro are replaced by `subst2`. Strings (for the purposes of this type of substitution) are delimited by blanks, tabs, new-line characters, and beginnings of lines. An example of the use of the substitute sequence is shown under **Libraries.**

Internal Macros

There are five internally maintained macros which are useful for writing rules for building targets.

`$*` The macro `$*` stands for the file name part of the current dependent with the suffix deleted. It is evaluated only for inference rules.

`$@` The `$@` macro stands for the full target name of the current target. It is evaluated only for explicitly named dependencies.

`$<` The `$<` macro is only evaluated for inference rules or the **.DEFAULT** rule. It is the module which is out-of-date with respect to the target (i.e., the "manufactured" dependent file name). Thus, in the **.c.o** rule, the `$<` macro would evaluate to

the `.c` file. An example for making optimized `.o` files from `.c` files is:

```
.c.o:
    cc -c -O $*.c
```

or:

```
.c.o:
    cc -c -O $<
```

$? The `$?` macro is evaluated when explicit rules from the makefile are evaluated. It is the list of prerequisites that are out-of-date with respect to the target; essentially, those modules which must be rebuilt.

$% The `$%` macro is only evaluated when the target is an archive library member of the form `lib(file.o)`. In this case, `$@` evaluates to `lib` and `$%` evaluates to the library member, `file.o`.

Four of the five macros can have alternative forms. When an upper case `D` or **F** is appended to any of the four macros, the meaning is changed to "directory part" for `D` and "file part" for `F`. Thus, `$(@D)` refers to the directory part of the string `$@`. If there is no directory part, `./` is generated. The only macro excluded from this alternative form is `$?`.

Suffixes

Certain names (for instance, those ending with `.o`) have inferable prerequisites such as `.c`, `.s`, etc. If no update commands for such a file appear in `makefile`, and if an inferable prerequisite exists, that prerequisite is compiled to make the target. In this case, `make` has inference rules which allow building files from other files by examining the suffixes and determining an appropriate inference rule to use. Inference rules in the makefile override the default rules.

The internal rules for `make` are compiled into the `make` program. To print out the rules compiled into the `make` program, the following command is used:

```
make -fp - 2>/dev/null </dev/null
```

A tilde in the above rules refers to an SCCS file. Thus, the rule **.c .o** would transform an SCCS C source file into an object file (**.o**). Because the **s.** of the SCCS files is a prefix, it is incompatible with `make`'s suffix point of view. Hence, the tilde is a way of changing any file reference into an SCCS file reference.

A rule with only one suffix (i.e., **.c:**) is the definition of how to build *x* from *x***.c**. In effect, the other suffix is null. This is useful for building targets from only one source file (e.g., command scripts, simple C programs).

Additional suffixes are given as the dependency list for **.SUFFIXES**. Order is significant; the first possible name for which both a file and a rule exist is inferred as a prerequisite.

Here again, the above command for printing the internal rules will display the list of suffixes implemented on the current machine. Multiple suffix lists accumulate; **.SUFFIXES:** with no dependencies clears the list of suffixes.

Inference Rules

The first example can be done more briefly.

```
pgm:  a.o  b.o
      cc  a.o  b.o  -o  pgm
a.o  b.o:  incl.h
```

This is because `make` has a set of internal rules for building files. The user may add rules to this list by simply putting them in the `makefile`.

Certain macros are used by the default inference rules to permit the inclusion of optional matter in any resulting commands. For example, `BCFLAGS`, `BLFLAGS`, and `BYFLAGS` are used for compiler options to `cc`, `lex`, and `yacc`, respectively. Again, the previous method for examining the current rules is recommended.

The inference of prerequisites can be controlled. The rule to create a file with suffix `.o` from a file with suffix `.c` is specified as an entry with **.c.o:** as the target and no dependents. Commands associated with the target define the rule for making a **.o** file from a **.c** file. Any target that has no slashes in it and starts with a dot is identified as a rule and not a true target.

Libraries

If a target or dependency name contains parentheses, it is assumed to be an archive library, the string within parentheses referring to a member within the library. Thus `lib(file.o)` and **$(LIB)(file.o)** both refer to an archive library which contains **file.o**. (This assumes the `LIB` macro has been previously defined.) The expression **$(LIB)(file1.o file2.o)** is not legal. Rules pertaining to archive libraries have the form *.XX***.a** where the *XX* is the suffix from which the archive member is to be made. The most common use of the archive interface follows. Here, we assume the source files are all C type source:

```
lib:  lib(file1.o) lib(file2.o) lib(file3.o)
      @echo lib is now up-to-date
.c.a:
      $(CC) -c $(CFLAGS) $<
      ar rv $@ $*.o
      rm -f $*.o
```

In fact, the **.c.a** rule listed above is built into `make` and is unnecessary in this example. A more interesting, but more limited example of an archive library maintenance construction follows:

```
lib:  lib(file1.o) lib(file2.o) lib(file3.o)
      $(CC) -c $(CFLAGS) $(?:.o=.c)
      ar rv lib $?
      rm $?  @echo lib is now up-to-date
.c.a:;
```

Here the substitution mode of the macro expansions is used. The **$?** list is defined to be the set of object file names (inside **lib**) whose C source files are out-of-date. The substitution mode translates the **.o** to **.c**. Note also, the disabling of the **.c.a:** rule, which would have created each object file, one by one. This particular construct speeds up archive library maintenance considerably. This type of construct becomes very cumbersome if the archive library contains a mix of assembly programs and C programs.

FILES

`[Mm]akefile and s.[Mm]akefile`

SEE ALSO

CC(SD_CMD), LEX(SD_CMD), SH(BU_CMD), YACC(SD_CMD).

USAGE

General.

The characters **= : @** in file names may give trouble.

LEVEL

Level 1.

NAME

nm – print name list of common object file

SYNOPSIS

`nm [ options ] file ...`

DESCRIPTION

The `nm` command displays the symbol table of each common object file `file`. The argument `file` may be a relocatable or absolute common object file; or it may be an archive of relocatable or absolute common object files. For each symbol, at least the following information will be printed:

Name The name of the symbol.

Value Its value expressed as an offset or an address depending on its storage class.

Size Its size in bytes, if available.

The output of `nm` may be controlled using the following options:

`-o` Print the value and size of a symbol in octal instead of decimal.

`-x` Print the value and size of a symbol in hexadecimal instead of decimal.

`-e` Print only external and static symbols.

`-f` Produce full output. Print redundant symbols (.text, .data and .bss), normally suppressed.

`-u` Print undefined symbols only.

`-V` Print the version of the nm command executing on the standard error output.

SEE ALSO

CC(SD_CMD), LD(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

prof — display profile data

SYNOPSIS

```
prof [-tcan] [-ox] [-g] [-z] [-m mdata] [prog]
```

DESCRIPTION

The command `prof` interprets a profile file produced by the `monitor` routine. The symbol table in the object file `prog` (`a.out` by default) is read and correlated with a profile file (`mon.out` by default). For each external text symbol the percentage of time spent executing between the address of that symbol and the address of the next is printed, together with the number of times that function was called and the average number of milliseconds per call.

The mutually exclusive options `t`, `c`, `a`, and `n` determine the type of sorting of the output lines:

`-t` Sort by decreasing percentage of total time (default).

`-c` Sort by decreasing number of calls.

`-a` Sort by increasing symbol address.

`-n` Sort lexically by symbol name.

The mutually exclusive options `o` and `x` specify the printing of the address of each symbol monitored:

`-o` Print each symbol address (in octal) along with the symbol name.

`-x` Print each symbol address (in hexadecimal) along with the symbol name.

The following options may be used in any combination:

`-g` Include non-global symbols (static functions).

`-z` Include all symbols in the profile range, even if associated with zero number of calls and zero time.

`-m` mdata
Use file `mdata` instead of `mon.out` as the input profile file.

A program creates a profile file if it has been loaded with the `-p` option of `cc` This option to the `cc` command arranges for calls to `monitor` at the beginning and end of execution. It is the call to `monitor` at the end of execution that causes a profile file to be written. The number of calls to a function is tallied if the `-p` option was used when the file containing the function was compiled.

The name of the file created by a profiled program is controlled by the environmental variable `PROFDIR` If `PROFDIR` is not set, "mon.out" is produced in the directory current when the program terminates. If `PROFDIR=string`, "`string/pid.progname`" is produced, where

`progname` consists of argv[0] with any path prefix removed, and `pid` is the program's process ID. If `PROFDIR` is set, but null, no profiling output is produced.

A single function may be split into subfunctions for profiling by means of the `MARK` macro [see MARK(SD_LIB)].

FILES

`mon.out` for profile
`a.out` for namelist

SEE ALSO

CC(SD_CMD), EXIT(BA_OS), PROFIL(KE_OS), MONITOR(SD_LIB), MARK(SD_LIB).

USAGE

General.

The times reported in successive identical runs may show variances, because of varying cache-hit ratios due to sharing of the cache with other processes. Even if a program seems to be the only one using the machine, hidden background or asynchronous processes may blur the data.

In rare cases, the clock ticks initiating recording of the program counter may "beat" with loops in a program, grossly distorting measurements. Call counts are always recorded precisely, however.

Only programs that call `exit` or return from `main` are guaranteed to produce a profile file, unless a final call to monitor is explicitly coded.

LEVEL

Level 1.

Optional. Requires the `profil` system service routine.

NAME

prs – print an SCCS file

SYNOPSIS

```
prs [ options ] files
```

DESCRIPTION

The command `prs` prints, on the standard output, parts or all of an SCCS file in a user-supplied format. If a directory is named, `prs` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**), and unreadable files are silently ignored. If a name of – is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file or directory to be processed; non-SCCS files and unreadable files are silently ignored.

Arguments to `prs`, which may appear in any order, consist of options, and file names.

All the described options apply independently to each named file. (Note that the `-c` option is new to System V Release 2.)

`-d`[*dataspec*] Used to specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see **DATA KEYWORDS**) interspersed with optional user supplied text. to specify the SCCS identification string of a delta for which information is desired. If no SID is specified, the SID of the most recently created delta is assumed.

`-r`SID Requests information for all deltas created *earlier* than and including the delta designated via the `-r` keyletter or the date given by the `-c` option.

`-l` Requests information for all deltas created *later* than and including the delta designated via the `-r` keyletter or the date given by the `-c` option.

`-c`[*date-time*] The cutoff *date-time* is in the form:

YY[MM[DD[HH[MM[SS]]]]]

Units omitted from the date-time default to their maximum possible values; for example, **-c7502** is equivalent to **-c750228235959**. Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date in the form: "**-c77/2/2 9:22:25**".

`-a` Requests printing of information for both removed, i.e., delta type = *R*, [see RMDEL(SD_CMD)] and existing, i.e., delta type = *D*, deltas. If the `-a` keyletter is not specified, information for existing deltas only is provided.

DATA KEYWORDS

Data keywords specify which parts of an SCCS file are to be retrieved and

output. All parts of an SCCS file have an associated data keyword. There is no limit on the number of times a data keyword may appear in a *dataspec*.

The information printed by `prs` consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognized data keywords in the order of appearance in the *dataspec*. The format of a data keyword value is either *Simple* (S), in which keyword substitution is direct, or *Multi-line* (M), in which keyword substitution is followed by a carriage return.

User-supplied text is any text other than recognized data keywords. A tab is specified by **\t** and carriage return/new-line is specified by **\n**. The default data keywords are:

":Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:"

TABLE 1. SCCS Files Data Keywords

Keyword	*Data Item*	*File Section*	*Value*	*Format*
:Dt:	Delta information	Delta Table	See below*	S
:DL:	Delta line statistics	"	:Li:/:Ld:/:Lu:	S
:Li:	Lines inserted by Delta	"	nnnnn	S
:Ld:	Lines deleted by Delta	"	nnnnn	S
:Lu:	Lines unchanged by Delta	"	nnnnn	S
:DT:	Delta type	"	*D* or *R*	S
:I:	SCCS ID string (SID)	"	:R:.:L:.:B:.:S:	S
:R:	Release number	"	nnnn	S
:L:	Level number	"	nnnn	S
:B:	Branch number	"	nnnn	S
:S:	Sequence number	"	nnnn	S
:D:	Date Delta created	"	:Dy:/:Dm:/:Dd:	S
:Dy:	Year Delta created	"	nn	S
:Dm:	Month Delta created	"	nn	S
:Dd:	Day Delta created	"	nn	S
:T:	Time Delta created	"	:Th::Tm::Ts:	S
:Th:	Hour Delta created	"	nn	S
:Tm:	Minutes Delta created	"	nn	S
:Ts:	Seconds Delta created	"	nn	S
:P:	Programmer who created Delta	"	logname	S
:DS:	Delta sequence number	"	nnnn	S
:DP:	Predecessor Delta seq-no.	"	nnnn	S
:DI:	Seq-no. of deltas incl., excl., ignored	"	:Dn:/:Dx:/:Dg:	S
:Dn:	Deltas included (seq #)	"	:DS: :DS:...	S
:Dx:	Deltas excluded (seq #)	"	:DS: :DS:...	S
:Dg:	Deltas ignored (seq #)	"	:DS: :DS:...	S
:MR:	MR numbers for delta	"	text	M
:C:	Comments for delta	"	text	M

:UN:	User names	User Names	text	M
:FL:	Flag list	Flags	text	M
:Y:	Module type flag	"	text	S
:MF:	MR validation flag	"	*yes* or *no*	S
:MP:	MR validation pgm name	"	text	S
:KF:	Keyword error/warning flag	"	*yes* or *no*	S
:KV:	Keyword validation string	"	text	S
:BF:	Branch flag	"	*yes* or *no*	S
:J:	Joint edit flag	"	*yes* or *no*	S
:LK:	Locked releases	"	:R:...	S
:Q:	User-defined keyword	"	text	S
:M:	Module name	"	text	S
:FB:	Floor boundary	"	:R:	S
:CB:	Ceiling boundary	"	:R:	S
:Ds:	Default SID	"	:I:	S
:ND:	Null delta flag	"	*yes* or *no*	S
:FD:	File descriptive text	Comments	text	M
:BD:	Body	Body	text	M
:GB:	Gotten body	"	text	M
:W:	A form of *what*(SD_CMD) string	N/A	:Z::M:\t:I:	S
:A:	A form of *what*(SD_CMD) string	N/A	:Z::Y: :M: :I::Z:	S
:Z:	*what*(SD_CMD) string delimiter	N/A	@(#)	S
:F:	SCCS file name	N/A	text	S
:PN:	SCCS file path name	N/A	text	S

* :Dt: = :DT: :I: :D: :T: :P: :DS: :DP:

EXAMPLES

prs −d"Users and/or user IDs for :F: are:\n:UN:" s.file

may produce on the standard output:

```
Users and/or user IDs for s.file are:
xyz
131
abc
```

prs −d"Newest delta for pgm :M:: :I: Created :D: By :P:" −r s.file

may produce on the standard output:

```
Newest delta for pgm main.c: 3.7 Created 77/12/1 By cas
```

As a *special case:*

prs s.file

may produce on the standard output:

```
D 1.1 77/12/1 00:00:00 cas 1 000000/00000/00000
MRs:
bl78-12345
bl79-54321
COMMENTS:
this is the comment line for s.file initial delta
```

for each delta table entry of the "D" type. The only keyletter argument allowed to be used with the *special case* is the `-a` keyletter.

SEE ALSO

ADMIN(SD_CMD), DELTA(SD_CMD), GET(SD_CMD), WHAT(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

rmdel — remove a delta from an SCCS file

SYNOPSIS

`rmdel -r`

DESCRIPTION

The command `rmdel` removes the delta specified by the SID from each named SCCS file. The delta to be removed must be the newest (most recent) delta in its branch in the delta chain of each named SCCS file. In addition, the SID specified must **not** be that of a version being edited for the purpose of making a delta (i.e., if a *p-file* [see GET(SD_CMD)] exists for the named SCCS file, the SID specified must **not** appear in any entry of the *p-file*).

If a directory is named, `rmdel` behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the path name does not begin with **s.**) and unreadable files are silently ignored. If a name of — is given, the standard input is read; each line of the standard input is taken to be the name of an SCCS file to be processed; non-SCCS files and unreadable files are silently ignored.

The restrictions on removal of a delta are: (1) the user who made a delta can remove it; (2) the owner of the file and directory can remove a delta.

SEE ALSO

DELTA(SD_CMD), GET(SD_CMD), PRS(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

sact — print current SCCS file editing activity

SYNOPSIS

```
sact files
```

DESCRIPTION

The command `sact` informs the user of any impending deltas to a named SCCS file. This situation occurs when `get -e` has been previously executed without a subsequent execution of `delta`. If a directory is named on the command line, `sact` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of `-` is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

The output for each named file consists of five fields separated by spaces.

Field 1 specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

Field 2 specifies the SID for the new delta to be created.

Field 3 contains the logname of the user who will make the delta (i.e., executed a `get` for editing).

Field 4 contains the date that `get -e` was executed.

Field 5 contains the time that `get -e` was executed.

SEE ALSO

DELTA(SD_CMD), GET(SD_CMD), UNGET(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

sdb — symbolic debugger

SYNOPSIS

```
sdb [ objfile [ corfile [ directory-list ] ] ]
```

DESCRIPTION

The command `sdb` is a symbolic debugger that can be used with C and Fortran77 (F77) programs. It may be used to examine their object files and core files and to provide a controlled environment for their execution.

The argument `objfile` is an executable program file which has been compiled with the `-g` (debug) option; if it has not been compiled with the `-g` option, or if it is not an executable file, the symbolic capabilities of `sdb` will be limited, but the file can still be examined and the program debugged. The default for `objfile` is `a.out`. The argument `corfile` is assumed to be a core image file produced after executing `objfile`; the default for `corfile` is `core`. The core file need not be present. A `-` in place of `corfile` will force `sdb` to ignore any core image file. The colon-separated list of directories (*directory-list*) is used to locate the source files used to build `objfile`.

It is useful to know that at any time there is a *current line* and *current file*. If `corfile` exists then they are initially set to the line and file containing the source statement at which the process terminated. Otherwise, they are set to the first line in `main()`. The current line and file may be changed with the source file examination commands.

By default, warnings are provided if the source files used in producing `objfile` cannot be found, or are newer than `objfile`.

Names of variables are written just as they are in C or F77. (The command `sdb` does not truncate names.) Variables local to a procedure may be accessed using the form *procedure:variable*. If no procedure name is given, the procedure containing the current line is used by default.

It is also possible to refer to structure members as *variable.member*, pointers to structure members as *variable->member* and array elements as *variable[number]*. Pointers may be dereferenced by using the form *pointer[0]*. Combinations of these forms may also be used. F77 common variables may be referenced by using the name of the common block instead of the structure name. Blank common variables may be named by the form *.variable*. A number may be used in place of a structure variable name, in which case the number is viewed as the address of the structure, and the template used for the structure is that of the last structure referenced by `sdb`. An unqualified structure variable may also be used with various commands. Generally, `sdb` will interpret a structure as a set of variables. Thus, `sdb` will display the values of all the elements of a structure when it is requested to display a structure. An exception to this interpretation occurs when displaying variable addresses. An entire structure does have an address, and it is this value `sdb` displays, not the addresses of individual

elements.

Elements of a multidimensional array may be referenced as *variable*[*number*][*number*]..., or as *variable*[*number,number,...*]. In place of *number,* the form *number;number* may be used to indicate a range of values, * may be used to indicate all legitimate values for that subscript, or subscripts may be omitted entirely if they are the last subscripts and the full range of values is desired. As with structures, `sdb` displays all the values of an array or of the section of an array if trailing subscripts are omitted. It displays only the address of the array itself or of the section specified by the user if subscripts are omitted. A multidimensional parameter in an F77 program cannot be displayed as an array, but it is actually a pointer, whose value is the location of the array. The array itself can be accessed symbolically from the calling function.

A particular instance of a variable on the stack may be referenced by using the form *procedure:variable,number.* All the variations mentioned in naming variables may be used. *Number* is the occurrence of the specified procedure on the stack, counting the top, or most current, as the first. If no procedure is specified, the procedure currently executing is used by default.

It is also possible to specify a variable by its address. All forms of integer constants which are valid in C may be used, so that addresses may be input in decimal, octal or hexadecimal.

Line numbers in the source program are referred to as *file-name:number* or *procedure:number.* In either case the number is relative to the beginning of the file. If no procedure or file name is given, the current file is used by default. If no number is given, the first line of the named procedure or file is used.

While a process is running under `sdb`, all addresses refer to the executing program.

Commands

The commands for examining data in the program are:

`t` Print a stack trace of the terminated or halted program.

`T` Print the top line of the stack trace.
Print the value of `variable` according to length `l` and format `m`. A numeric count `c` indicates that a region of memory, beginning at the address implied by `variable`, is to be displayed. The length specifiers are:

`b` one byte

`h` two bytes (half word)

`l` four bytes (long word)

Legal values for **m** are:

- **c** character
- **d** decimal
- **u** decimal, unsigned
- **o** octal
- **x** hexadecimal
- **s** Assume *variable* is a string pointer and print characters starting at the address pointed to by the variable.
- **a** Print characters starting at the variable's address. This format may not be used with register variables.
- **p** pointer to procedure
- **i** disassemble machine-language instruction with addresses printed numerically and symbolically.

The length specifiers are only effective with the formats **c**, **d**, **u**, **o** and **x**. Any of the specifiers, *c*, *l*, and *m*, may be omitted. If all are omitted, `sdb` choses a length and a format suitable for the variable's type as declared in the program. If *m* is specified, then this format is used for displaying the variable. A length specifier determines the output length of the value to be displayed, sometimes resulting in truncation. A count specifier *c* tells `sdb` to display that many units of memory, beginning at the address of *variable*. The number of bytes in one such unit of memory is determined by the length specifier *l*, or if no length is given, by the size associated with the *variable*. If a count specifier is used for the **s** or **a** command, then that many characters are printed. Otherwise successive characters are printed until either a null byte is reached or 128 characters are printed. The last variable may be redisplayed with the command **./**.

linenumber?*lm*
variable:?*lm*

Print the value at the address from **a.out** or I space given by *linenumber* or *variable* (procedure name), according to the format *lm*. The default format is 'i'.

variable=*lm*
linenumber=*lm*
number=*lm*

Print the address of *variable* or *linenumber*, or the value of *number*, in the format specified by *lm*. If no format is given, then **lx** is used. The last variant of this command provides a convenient way to convert between decimal, octal and hexadecimal.

variable!*value*

Set *variable* to the given *value*. The value may be a number, a character constant or a variable. The value must be well defined; expressions which produce more than one value, such as structures, are not allowed. Character constants are denoted *'character*. Numbers are viewed as integers unless a decimal point or

exponent is used. In this case, they are treated as having the type double. Registers are viewed as integers. The *variable* may be an expression which indicates more than one variable, such as an array or structure name. If the address of a variable is given, it is regarded as the address of a variable of type *int*. C conventions are used in any type conversions necessary to perform the indicated assignment.

x Print the machine registers and the current machine-language instruction.

The commands for examining source files are:

e *procedure*
e *file-name*
e *directory/*
e *directory file-name*
The first two forms set the current file to the file containing *procedure* or to *file-name*. The current line is set to the first line in the named procedure or file. Source files are assumed to be in *directory*. The default is the current working directory. The latter two forms change the value of *directory*. If no procedure, file name, or directory is given, the current procedure name and file name are reported.

/regular expression/
Search forward from the current line for a line containing a string matching *regular expression* as in ED(BU_CMD). The trailing **/** may be deleted.

?regular expression?
Search backward from the current line for a line containing a string matching *regular expression* as in ED(BU_CMD). The trailing **?** may be deleted.

p Print the current line.

z Print the current line followed by the next 9 lines. Set the current line to the last line printed.

w Window. Print the 10 lines around the current line.

number
Set the current line to the given line number. Print the new current line.

The commands for controlling the execution of the source program are:

count **r** *args*
count **R**
Run the program with the given arguments. The **r** command with no arguments reuses the previous arguments to the program while the **R** command runs the program with no arguments. An argument beginning with < or > causes redirection for the

standard input or output, respectively. If *count* is given, it specifies the number of breakpoints to be ignored.

linenumber **c** *count*
linenumber **C** *count*

Continue after a breakpoint or interrupt. If *count* is given, the program will stop when *count* breakpoints have been encountered. With the **C** command, the signal which caused the program to stop is reactivated; with the **c** command, it is ignored. If a line number is specified then a temporary breakpoint is placed at the line and execution is continued. The breakpoint is deleted when the command finishes. (It may not be possible to set breakpoints in some places, e.g., with shared libraries.)

s *count*
S *count*

Single step the program through *count* lines. If no count is given then the program is run for one line. **S** is equivalent to **s** except it steps through procedure calls.

i
I

Single step by one machine-language instruction. With the **I** command, the signal which caused the program to stop is reactivated; with the **i** command, it is ignored.

k Kill the program being debugged.

procedure(arg1,arg2,...)
procedure(arg1,arg2,...)*/m*

Execute the named procedure with the given arguments. Arguments can be integer, character or string constants or names of variables accessible from the current procedure. The second form causes the value returned by the procedure to be printed according to format *m*. If no format is given, it defaults to **d.**

linenumber **b** *commands*

Set a breakpoint at the given line. If a procedure name without a line number is given (e.g., "proc:"), a breakpoint is placed at the first line in the procedure even if it was not compiled with the **−g** option. If no *linenumber* is given, a breakpoint is placed at the current line. (It may not be possible to set breakpoints in some places, e.g., with shared libraries.)

If no *commands* are given, execution stops just before the breakpoint and control is returned to `sdb`. Otherwise the *commands* are executed when the breakpoint is encountered and execution continues. Multiple commands are specified by separating them with semicolons. If **k** is used as a command to execute at a breakpoint, control returns to `sdb`, instead of continuing execution.

B Print a list of the currently active breakpoints.

linenumber **d**
Delete a breakpoint at the given line. If no *linenumber* is given then the breakpoints are deleted interactively. Each breakpoint location is printed and a line is read from the standard input. If the line begins with a **y** or **d** then the breakpoint is deleted.

D Delete all breakpoints.

l Print the last executed line.

Miscellaneous commands:

!*command*
The command is interpreted by the command interpreter.

new-line
If the previous command printed a source line, then advance the current line by one line and print the new current line. If the previous command displayed a memory location, then display the next memory location.

end-of-file
Scroll. Print the next 10 lines of instructions, source or data depending on which was printed last. (The end-of-file character is usually control-D.)

< *filename*
Read commands from *filename* until the end of file is reached, and then continue to accept commands from standard input. When `sdb` is told to display a variable by a command in such a file, the variable name is displayed along with the value. This command may not be nested; < may not appear as a command in a file.

" *string*
Print the given string. The C escape sequences of the form *character* are recognized, where *character* is a nonnumeric character.

q Exit the debugger.

FILES
a.out
core

SEE ALSO
CC(SD_CMD), ED(BU_CMD).

USAGE
General.

LEVEL
Level 1.

NAME

size — print section sizes of object files

SYNOPSIS

```
size [-o] [-x] [-V] files
```

DESCRIPTION

The *size* command produces section size information for each section in the loaded object files. The sizes of the loaded sections are printed along with the sum of these sizes. If an archive file is input to the *size* command, the information for all archive members is displayed.

Numbers will be printed in decimal unless either the **-o** or the **-x** option is used, in which case they will be printed in octal or in hexadecimal, respectively.

The **-V** flag will supply the version information on the *size* command.

SEE ALSO

CC(SD_CMD), LD(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

strip – strip symbolic information from an object file

SYNOPSIS

```
strip [-x] [-r] [-V] file ...
```

DESCRIPTION

The *strip* command strips the symbolic information from object files or archives of object files.

The amount of information stripped from the symbol table can be controlled by using any of the following options:

-x Do not strip static or external symbol information.

-r Do not strip static or external symbol information, or relocation information.

-V Print the version of the strip command, on the standard error output.

If there is any relocation information in the object file and any symbol table information is to be stripped, *strip* will report an error and terminate without stripping *file* unless the **-r** flag is used.

SEE ALSO

AR(BU_CMD), CC(SD_CMD), LD(SD_CMD).

USAGE

General.

The purpose of this command is to reduce the file storage overhead taken by the object file.

LEVEL

Level 1.

NAME

time — time a command

SYNOPSIS

`time command`

DESCRIPTION

The `command` is executed; after it is complete, `time` prints the elapsed time during the command, the time spent executing system code, and the time spent in execution of the user code. Times are reported in seconds.

The times are printed on standard error.

USAGE

General.

When `time` is used on a multi-processor system the sum of system and user time could be greater than real time.

LEVEL

Level 1.

NAME

tsort — topological sort

SYNOPSIS

```
tsort [ file ]
```

DESCRIPTION

`Tsort` produces on the standard output a totally ordered list of items consistent with a partial ordering of items mentioned in the input `file`. If no `file` is specified, the standard input is understood.

The input consists of pairs of items (nonempty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

SEE ALSO

LORDER(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

unget — undo a previous get of an SCCS file

SYNOPSIS

```
unget [-rSID] [-s] [-n] files
```

DESCRIPTION

`Unget` undoes the effect of a `get -e` done prior to creating the intended new delta. If a directory is named, `unget` behaves as though each file in the directory were specified as a named file, except that non-SCCS files and unreadable files are silently ignored. If a name of `-` is given, the standard input is read with each line being taken as the name of an SCCS file to be processed.

Keyletter arguments apply independently to each named file.

`-r`*SID* Uniquely identifies which delta is no longer intended. (This would have been specified by `get` as the new delta). The use of this keyletter is necessary only if two or more outstanding `gets` for editing on the same SCCS file were done by the same person (login name). An error is reported if the specified *SID* is ambiguous, or if it is necessary and omitted on the command line.

`-s` Suppresses the printout, on the standard output, of the intended delta's *SID*.

`-n` Causes the retention of the file that was obtained by `get`, which would normally be removed from the current directory.

SEE ALSO

DELTA(SD_CMD), GET(SD_CMD), SACT(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

val — validate SCCS file

SYNOPSIS

```
val -
val [ -s ] [ -rSCCS ] [ -mname ] [ -ytype ] file
...
```

DESCRIPTION

The command `val` determines if the specified `file` is an SCCS file meeting the characteristics specified by the options. The arguments may appear in any order.

`val` has a special argument, `-`, which causes reading of the standard input until an end-of-file condition is detected. Each line read is independently processed as if it were a command line argument list.

`val` generates diagnostic messages on the standard output for each command line and file processed, and also returns a single 8-bit code upon exit as described below.

The options are defined as follows. The effects of any option apply independently to each named file on the command line.

`-s` Silences the diagnostic message normally generated on the standard output for any error that is detected while processing each named file on a given command line.

`-r`*SID* *SID* (SCCS Identification String) is an SCCS delta number. A check is made to determine if the *SID* is ambiguous (e. g., `-r1` is ambiguous because it physically does not exist but implies 1.1, 1.2, etc., which may exist) or invalid (e. g., `-r1.0` or `-r1.1.0` are invalid because neither case can exist as a valid delta number). If the *SID* is valid and not ambiguous, a check is made to determine if it actually exists.

`-m`*name* `name` is compared with the SCCS %M% keyword in `file`.

`-y`*type* `type` is compared with the SCCS %Y% keyword in `file`.

The 8-bit code returned by `val` is a disjunction of the possible errors, i. e., can be interpreted as a bit string where (moving from left to right) set bits are interpreted as follows:

bit 0 = missing file argument;
bit 1 = unknown or duplicate keyletter argument;
bit 2 = corrupted SCCS file;
bit 3 = cannot open file or file not SCCS;
bit 4 = *SID* is invalid or ambiguous;
bit 5 = *SID* does not exist;
bit 6 = %Y%, **-y** mismatch;
bit 7 = %M%, **-m** mismatch;

Note that `val` can process two or more files on a given command line and in turn can process multiple command lines (when reading the standard input). In these cases an aggregate code is returned — a logical **OR** of the codes generated for each command line and file processed.

SEE ALSO

ADMIN(SD_CMD), DELTA(SD_CMD), GET(SD_CMD), PRS(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

what — identify SCCS files

SYNOPSIS

```
what [-s] files
```

DESCRIPTION

The command `what` searches the given files for all occurrences of the pattern that the `get` command substitutes for %Z% (@(#)) and prints out what follows until the first ", >, new-line, \, or null character. For example, if the C program in file **f.c** contains

```
char ident[] = "@(#)identification informa-
tion";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command

```
what f.c f.o a.out
```

will print

f.c: identification information

f.o: identification information

a.out: identification information

`What` is intended to be used in conjunction with the SCCS `get` command, which automatically inserts identifying information, but it can also be used where the information is inserted manually.

There is only one option (new in System V Release 2):

`-s` Quit after finding the first occurrence of pattern in each file.

ERRORS

Exit status is 0 if any matches are found, otherwise 1.

SEE ALSO

GET(SD_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

xargs — construct argument list(s) and execute command

SYNOPSIS

`xargs [options] [command[initial-arguments]]`

DESCRIPTION

`Xargs` combines the fixed `initial-arguments` with arguments read from standard input to execute the specified `command` one or more times. The number of arguments read for each `command` invocation and the manner in which they are combined are determined by the options specified.

If `command` is omitted, `echo` is used.

Arguments read in from standard input are defined to be contiguous strings of characters delimited by one or more blanks, tabs, or new-lines; empty lines are always discarded. Blanks and tabs may be embedded as part of an argument if escaped or quoted. Characters enclosed in quotes (single or double) are taken literally, and the delimiting quotes are removed. Outside of quoted strings a backslash (\) quotes the next character.

Each argument list is constructed starting with the `initial-arguments`, followed by some number of arguments read from standard input (Exception: see `-i`). Options `-i`, `-l`, and `-n` determine how arguments are selected for each command invocation. When none of these options are coded, the `initial-arguments` are followed by arguments read continuously from standard input until an internal buffer is full, and then `command` is executed with the accumulated args. This process is repeated until there are no more args. When there are conflicts (e.g., `-l` vs. `-n`), the last option has precedence. The recognized options are:

`-l`*number* `Command` is executed for each non-empty `number` lines of arguments from standard input. The last invocation of `command` will be with fewer lines of arguments if fewer than `number` remain. A line is considered to end with the first new-line `unless` the last character of the line is a blank or a tab; a trailing blank/tab signals continuation through the next non-empty line. If `number` is omitted, 1 is assumed. Option `-x` is forced.

`-i`*replstr* Insert mode: `command` is executed for each line from standard input, taking the entire line as a single arg, inserting it in `initial-arguments` for each occurrence of `replstr`. A maximum of 5 arguments in `initial-arguments` may each contain one or more instances of `replstr`. Blanks and tabs at the beginning of each line are thrown away. Constructed arguments may not grow larger than 255 characters, and option `-x` is also forced. {} is assumed for `replstr` if not specified.

−n*number* Execute `command` using as many standard input arguments as possible, up to `number` arguments maximum. Fewer arguments will be used if their total size is greater than `size` characters, and for the last invocation if there are fewer than `number` arguments remaining. If option `−x` is also invoked, each `number` arguments must fit in the `size` limitation, else `xargs` terminates execution.

−t Trace mode: The `command` and each constructed argument list are echoed to standard error just prior to their execution.

−p Prompt mode: The user is asked whether to execute `command` each invocation. Trace mode (`−t`) is turned on to print the command instance to be executed, followed by a `?...` prompt. A reply of `y` (optionally followed by anything) will execute the command; anything else, including just a carriage return, skips that particular invocation of `command`.

−x Causes `xargs` to terminate if any argument list would be greater than `size` characters; `−x` is forced by the options `−i` and `−l`. When neither of the options `−i`, `−l`, or `−n` are coded, the total length of all arguments must be within the `size` limit.

−s*size* The maximum total size of each argument list is set to `size` characters; `size` must be a positive integer less than or equal to 470. If `−s` is not coded, 470 is taken as the default. Note that the character count for `size` includes one extra character for each argument and the count of characters in the command name.

−e*eofstr* *Eofstr* is taken as the logical end-of-file string. Underscore (_) is assumed for the logical EOF string if `−e` is not invoked. The option `−e` with no `eofstr` coded turns off the logical EOF string capability (underbar is taken literally). `Xargs` reads standard input until either end-of-file or the logical EOF string is encountered.

`Xargs` will terminate if either it receives a return code of `−1` from, or if it cannot execute, `command`. (Thus `command` should explicitly `exit` with an appropriate value to avoid accidentally returning with `−1`.)

EXAMPLES

The following will move all files from directory $1 to directory $2, and echo each move command just before doing it:

```
ls $1 | xargs −i −t mv $1/{} $2/{}
```

The following will combine the output of the parenthesized commands onto one line, which is then echoed to the end of file `log`:

```
(logname; date; echo $0 $*) | xargs >>log
```

The user is asked which files in the current directory are to be archived and archives them into `arch` (1.) one at a time, or (2.) many at a time.

```
1. ls | xargs -p -1 ar r arch
2. ls | xargs -p -1 | xargs ar r arch
```

The following will execute with successive pairs of arguments originally typed as command line arguments:

```
echo $* | xargs -n2 diff
```

SEE ALSO

ECHO(BU_CMD).

USAGE

General.

LEVEL

Level 1.

NAME

yacc — a compiler-compiler

SYNOPSIS

`yacc [ -vdlt ] grammar`

DESCRIPTION

The `yacc` command provides a general tool for describing the input to a program. More precisely, `yacc` converts a context-free grammar into a set of tables for a simple automaton which executes an LR(1) parsing algorithm. The grammar may be ambiguous; built-in precedence rules are used to break ambiguities.

The output file, **y.tab.c,** must be compiled by the C compiler to produce a program `yyparse`. This program must be loaded with the lexical analyzer function, `yylex`, as well as `main` and `yyerror`, an error handling routine. These routines must be supplied by the user (however, see the description of the `yacc` library below); `lex` is useful for creating lexical analyzers usable by `yacc`.

If the `-v` option is used, the file `y.output` is prepared, which contains a description of the parsing tables and a report on conflicts generated by ambiguities in the grammar.

If the `-d` option is used, the file `y.tab.h` is generated with the **#define** statements that associate the `yacc`-assigned "token codes" with the user-declared "token names". This allows source files other than **y.tab.c** to access the token codes.

If the `-l` option is used, the code produced in **y.tab.c** will *not* contain any **#line** constructs. This should only be used after the grammar and the associated actions are fully debugged.

Runtime debugging code is always generated in **y.tab.c** under conditional compilation control. By default, this code is not included when **y.tab.c** is compiled. However, when `yacc`'s `-t` option is used, this debugging code will be compiled by default. Independent of whether the `-t` option was used, the runtime debugging code is under the control of **YYDEBUG,** a preprocessor symbol. If **YYDEBUG** has a non-zero value, then the debugging code is included. If its value is zero, then the code will not be included. The size and execution time of a program produced without the runtime debugging code will be smaller and slightly faster.

Yacc Library

The `yacc` library **liby.a** facilitates the initial use of `yacc` by providing the routines:

```
main()

yyerror(s)
  char *s;
```

These routines may be loaded by using the `-ly` option with `cc`. **main()** just calls **yyparse()**. **yyerror()** simply prints the string (error

message) s when a syntax error is detected.

YACC SPECIFICATIONS

The `yacc` user constructs a specification of the input process; this includes rules describing the input structure, code to be invoked when these rules are recognized, and a low-level routine to do the basic input. `Yacc` then generates the (integer-valued) function `yyparse`; it in turn calls `yylex`, the lexical analyzer, to obtain input tokens.

A structure recognized (and returned) by the lexical analyzer is called a *terminal symbol*, here referred to as a `token` (literal characters must also be passed through the lexical analyzer, and are also considered tokens). A structure recognized by the parser is called a *nonterminal symbol*. `Name` refers to either tokens or nonterminal symbols.

Every specification file consists of three sections: `declarations`, `grammar rules`, and `programs`, separated by double percent marks ("%%"). The declarations and programs sections may be empty. If the latter is empty, then the preceding %% mark separating it from the rules section may be omitted.

Blanks, tabs and newlines are ignored, except that they may not appear in names or multi-character reserved symbols. Comments are enclosed in /* ... */, and may appear wherever a name is legal.

Names may be of arbitrary length, made up of letters, dot ".", underscore "_", and non-initial digits. Upper and lower case letters are distinct. Names beginning in "yy" should be avoided, since the `yacc` parser uses such names.

A literal consists of a character enclosed in single quotes. The C escape sequences (e.g., '\n') are recognized.

Declarations

The following declarators may be used in the declarations section:

%token

Names representing tokens must be declared; this is done by writing

```
%token name1 name2 ...
```

in the declarations section. Every name not defined in this section is assumed to represent a nonterminal symbol. Every nonterminal symbol must appear on the left side of at least one grammar rule.

%start

The `start symbol` represents the largest, most general structure described by the grammar rules. By default, it is the left hand side of the first grammar rule; this default may be overridden by declaring:

```
%start symbol
```

%left

%right

%nonassoc

Precedence and associativity rules attached to tokens are declared using these keywords. This is done by a series of lines, each beginning with one of the keywords %left, %right, or %nonassoc, followed by a list of tokens. All tokens on the same line have the same precedence level and associativity; the lines are in order of increasing precedence or binding strength. %left denotes that the operators on that line are left associative, and %right similary denotes right associative operators. %nonassoc denotes operators that may not associate with themselves. (A token declared using one of these keywords need not be declared by %token as well.)

%prec

Unary operators must, in general, be given a precedence. In cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, the keyword %prec is used to change the precedence level associated with a particular grammar rule. %prec appears immediately after the body of the grammar rule, before the action or closing semicolon (see Grammar Rules below), and is followed by a token name or a literal. It causes the precedence of the grammar rule to become that of the following token name or literal.

%union

By default, the values returned by actions and the lexical analyzer are integers. Other types, including structures, are supported: the `yacc` value stack is declared to be a union of the various types of values desired. `Yacc` keeps track of types, and inserts appropriate union member names so that the resulting parser will be strictly type-checked. The declaration is done by including a statement of the form:

```
%union {
        body of union
}
```

Alternatively, the union may be declared in a header file, and a typedef used to define the variable YYSTYPE to represent this union. The header file must be included in the declarations section, by using a "#include" construct within %{ and %} (see below). Union members must be associated with the various names. The construction <name> is used to indicate a union member name; if this follows one of the keywords %token %left, %right, and %nonassoc, the union member name is associated with the tokens listed.

%type

This key word is used to associate union member names with

nonterminals, in the form:

```
%type <ntype> a b ...
```

Other declarations and definitions can appear in the declarations section, enclosed by the marks "%{" and "%}". These have global scope within the file, so that they may be used in the rules and programs sections.

Grammar Rules

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

A represents a nonterminal name, and BODY represents a sequence of zero or more names and literals. The colon and the semicolon are `yacc` punctuation. If there are several successive grammar rules with the same left hand side, the vertical bar '|' can be used to avoid rewriting the left hand side; in this case the semicolon must occur only after the last rule. The BODY part may be empty to indicate that the nonterminal symbol matches the empty string.

The ASCII NUL character (0 or '\0') should not be used in grammar rules.

With each grammar rule, the user may associate actions to be performed each time the rule is recognized in the input process. These actions may return values, and may obtain the values returned by previous actions. In addition, the lexical anlayzer can return values for tokens, if desired.

An action is an arbitrary C statement, and as such can do input or output, call subprograms, and alter external variables. An action is one or more statements enclosed in curly braces "{" and "}". Certain pseudo-variables can be used in the action: a value can be returned by assigning it to $$; the variables $1, $2, ..., refer to the values returned by the components of the right side of a rule, reading from left to right. By default, the value of a rule is the value of the first element in it. Actions may occur in the middle of a rule as well as at the end; an action may access the values returned by symbols (and actions) to its left, and in turn the value it returns may be accessed by actions to its right.

Internal rules to resolve ambiguities are:

1. In a shift/reduce conflict, the default is to do the shift.
2. In a reduce/reduce conflict, the default is to reduce by the grammar rule that occurs *earlier* in the input sequence.

In addition, the declared precedences and associativities (see Declarations Section above) are used to resolve parsing conflicts as follows:

1. A precedence and associativity is associated with each grammar rule; it is the precedence and associativity of the last token or literal in the body of the rule. If the %prec keyword is used, it overrides this default. Some grammar rules may have no precedence and associativity.

2. When there is a reduce/reduce conflict, or there is a shift/reduce conflict and either the input symbol or the grammar rule has no precedence and associativity, then the two rules given above are used.

3. If there is a shift/reduce conflict, and both the grammar rule and the input symbol have precedence and associativity associated with them, then the conflict is resolved in favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies error.

Conflicts resolved by precedence are not counted in the shift/reduce and reduce/reduce conflicts reported by `yacc`.

The token name "error" is reserved for error handling. This name can be used in grammar rules; in effect, it suggests places where errors are expected, and recovery might take place. When an error is encountered, the parser behaves as if the token "error" were the current lookahead token, and performs the action encountered. The lookahead token is then reset to the token that caused the error. If no special error rules have been specified, the processing halts when an error is detected.

In order to prevent a series of error messages, the parser, after detecting an error, remains in error state until three tokens have been successfully read and shifted. If an error is detected when the parser is already in error state, no message is given, and the input token is quietly deleted.

The statement

```
yyerrok;
```

in an action resets the parser back to its normal mode; it may be used if it is desired to force the parser to believe that an error has been fully recovered from.

The statement

```
yyclearin;
```

in an action is used to clear the previous lookahead token; it may be used if a user-supplied routine is to be used to find the correct place to resume input.

Programs

The programs section may include the definition of the lexical analyzer `yylex`, and any other functions, for example those used in the actions

specified in the grammar rules.

`yylex` is an integer-valued function, which returns the *token number*, representing the kind of token read. If there is a value associated with that token, it should be assigned to the external variable `yylval`. The parser and `yylex` must agree on these token numbers in order for communication between them to take place. The numbers may be chosen by `yacc`, or chosen by the user. In either case, the "#define" construct of C is used to allow `yylex` to return these numbers symbolically. If the token numbers are chosen by `yacc`, then literals are given the numerical value of the character in the local character set, and other names are assigned token numbers starting at 257.

A token may be assigned a number by following its first appearance in the declarations section with a nonnegative integer. Names and literals not defined this way retain their default definition. All token numbers must be distinct.

The end of the input is marked by a special token called the `end-marker`. The endmarker must have token number 0 or negative. (these values are not legal for any other token.) All lexical analyzers should return 0 or negative as a token number upon reaching the end of their input. If the token upto, but excluding, the endmarker form a structure which matches the start symbol, the parser `accepts` the input. If the endmarker is seen in any other context, it is an error.

ERRORS

The number of reduce-reduce and shift-reduce conflicts is reported on the standard error output; a more detailed report is found in the **y.output** file. Similarly, if some rules are not reachable from the start symbol, this is also reported.

FILES

y.output
y.tab.c
y.tab.h

SEE ALSO

LEX(SD_CMD).

USAGE

General.

LEVEL

Level 1.

Part VI

Terminal Interface Extension Definition

Chapter 12
Introduction

12.1 OVERVIEW

The Terminal Interface Extension (TI) consists of the facilities provided by the *curses/terminfo* package to allow application programs to perform terminal-handling functions in a way that is independent of the type of the terminal actually in use. Currently, the *curses/terminfo* package supports asynchronous character terminals.

The System V Base, the Basic Utilities Extension, the Advanced Utilities Extension, and the Software Development Extension are prerequisites for the Terminal Interface Extension.

The components of the Terminal Interface Extension are new in System V Release 2.

12.2 DESCRIPTION

DATA FILES

/usr/lib/terminfo/?/*

UTILITIES

tic tput

LIBRARY ROUTINES

General Routines

addch	getyx	mvwinsch	standend
addstr	has_ic	mvwprintw	standout
attroff	has_il	mvwscanw	subwin
attron	idlok	newpad	touchwin
attrset	inch	newterm	typeahead
baudrate	initscr	newwin	unctrl
beep	insch	nl	waddch
box	insertln	nocbreak	waddstr
cbreak	intrflush	nodelay	wattroff
clear	keypad	noecho	wattron
clearok	killchar	nonl	wattrset
clrtobot	leaveok	noraw	wclear
clrtoeol	longname	overlay	wclrtobot
def_prog_mode	move	overwrite	wclrtoeol
def_shell_mode	mvaddch	pnoutrefresh	wdelch
delay_output	mvaddstr	prefresh	wdeleteln
delch	mvdelch	printw	werase
deleteln	mvgetch	raw	wgetch
delwin	mvgetstr	refresh	wgetstr
doupdate	mvinch	reset_prog_mode	winch
echo	mvinsch	reset_shell_mode	winsch
endwin	mvprintw	resetterm **	winsertln
erase	mvscanw	resetty	wmove
erasechar	mvwaddch	saveterm **	wnoutrefresh
fixterm **	mvwaddstr	savetty	wprintw
flash	mvwdelch	scanw	wrefresh
flushinp	mvwgetch	scroll	wscanw
getch	mvwgetstr	scrollok	wsetscrreg
getstr	mvwin	set_term	wstandend
gettmode **	mvwinch	setscrreg	wstandout

Terminfo Level Routines

mvcur	setterm **	tparm	vidattr
putp	setupterm	tputs	vidputs

Termcap Compatibility Routines

tgetent **	tgetnum **	tgoto **	tputs
tgetflag **	tgetstr **		

** Level 2: December 1, 1985.

12.3 DEFINITIONS

The following environment varirables are used by the components of the TI extension. See SH(BU_CMD) for information on the shell environment.

`TERM`
The environmental variable `TERM` usually contains a user's current terminal type and can be set by the user.

`TERMINFO`
The environmental variable `TERMINFO`, if set, contains the place(s) where local terminal descriptions can be found. `TERMINFO` can be set by the user. If it is set, any program using CURSES(TI_LIB) will check the `TERMINFO` location for a terminal's description before checking `/usr/lib/terminfo`, the standard location for terminal descriptions. See CURSES(TI_LIB) for further information.

`LINES` and `COLUMNS`
The environmental variables `LINES` and `COLUMNS`, if set, contain the number of lines and number of columns, respectively, on a terminal screen and can be set by the user. If defined, the values of these variables, `LINES` and `COLUMNS`, will override the screen size values given in a terminal's *terminfo* description. See CURSES(TI_LIB) for further information.

12.4 TRADEMARKS

Tektronix is a registered trademark of Tektronix, Inc.
TeleVideo is a registered trademark of TeleVideo Systems, Inc.
VT100 is a trademark of Digital Equipment Corporation.
LSI is a trademark of Lear Siegler, Inc.
HP is a trademark of Hewlett-Packard Co.
Tektronix 4010 is a registered trademark of Tektronic, Inc.
Beehive is a trademark of Beehive International.
Ann Arbor is a trademark of Ann Arbor Terminals, Inc.
Teleray is a trademark of Research, Inc.
Micro-Term, ACT, and MIME are trademarks of Micro-Term, Inc.
Concept is a trademark of Human Designed Systems, Inc.
Teletype is a trademark of AT&T Teletype Corporation.

Chapter 13
Environment

NAME

terminfo — terminal capability database

SYNOPSIS

```
/usr/lib/terminfo/?/*
```

DESCRIPTION

The *terminfo* database describes terminals, by giving a set of capabilities which they have, by describing how operations are performed, by describing padding requirements, and by specifying initialization sequences.

Entries in *terminfo* consist of a number of comma-separated fields. White space after each comma is ignored. The first entry for each terminal gives the names which are known for the terminal, separated by vertical bar (|) characters. The first name given is the most common abbreviation for the terminal, the last name given should be a long name fully identifying the terminal, and all others are understood as synonyms for the terminal name. All names but the last should be in lower case and contain no blanks; the last name may well contain upper case and blanks for readability.

Terminal names (except for the last, verbose entry) should be chosen using the following conventions. The particular piece of hardware making up the terminal should have a root name chosen, for example, "tty4424". Modes that the hardware can be in, or user preferences, should be indicated by appending a hyphen and an indicator of the mode. The following suffixes should be used where possible:

Suffix	Meaning
`-w`	Wide mode (more than 80 columns)
`-am`	With auto. margins (usually default)
`-nam`	Without automatic margins
-n	Number of lines on the screen (e.g., -60)
`-na`	No arrow keys (leave them in local)
-n`p`	Number of pages of memory (e.g., -8p)
`-rv`	Reverse video

To avoid conflicts with the naming conventions used in describing the different modes of a terminal (e.g., `-w`), it is recommended that a terminal's root name not contain hyphens. Further, it is good practice to make all terminal names used in the *terminfo* database unique.

Capabilities

In the table below, "**Variable**" is the name by which the programmer (at the *terminfo* level) accesses the capability. "**Capname**" is the short name used in the text of the database, and is used by a person updating the database. The "**Termcap Code**" is the two letter code that corresponds to the old *termcap* capability name.

Capability names have no hard length limit, but an informal limit of 5 characters has been adopted to keep them short. Whenever possible, names are

chosen to be the same as or similar to the ANSI X3.64-1979 standard. Semantics are also intended to match those of the specification.

All string capabilities listed below may have padding specified, with the exception of those used for input. Input capabilities, listed under the **Strings** section of the table below, are denoted by the string **key_** at the beginning of their variable name.

(G) indicates that the string is passed through `tparm()` with parms as given (*#i*).

(*) indicates that padding may be based on the number of lines affected

($\#_i$) indicates the i^{th} parameter.

Variable	Cap-name	Term-cap Code	Description
Booleans:			
auto_left_margin	bw	bw	**cub1** wraps from col 0 to last column
auto_right_margin	am	am	Terminal has automatic margins
(Reserved)	xsb	xb	Beehive (f1=escape, f2=ctrl C)
ceol_standout_glitch	xhp	xs	Standout not erased by overwriting
eat_newline_glitch	xenl	xn	Newline ignored after 80 cols
erase_overstrike	eo	eo	Can erase overstrikes with a blank
generic_type	gn	gn	Generic line type(e.g. dialup, switch)
hard_copy	hc	hc	Hardcopy terminal
has_meta_key	km	km	(Reserved)
has_status_line	hs	hs	Has extra "status line"
insert_null_glitch	in	in	Insert mode distinguishes nulls
memory_above	da	da	Display may be retained above screen
memory_below	db	db	Display may be retained below screen
move_insert_mode	mir	mi	Safe to move while in insert mode
move_standout_mode	msgr	ms	Safe to move in standout modes
over_strike	os	os	Terminal overstrikes
status_line_esc_ok	eslok	es	Escape can be used on the status line
(Reserved)	xt	xt	Destructive tabs, magic **smso** char
tilde_glitch	hz	hz	Cannot print tildes
transparent_underline	ul	ul	Underline character overstrikes
xon_xoff	xon	xo	Terminal uses xon/xoff handshaking
Numbers:			
columns	cols	co	Number of columns in a line
init_tabs	it	it	Tabs initially every # spaces
lines	lines	li	Number of lines on screen or page
lines_of_memory	lm	lm	Lines of memory if > **lines**, 0=varies
magic_cookie_glitch	xmc	sg	# of blank chars left by **smso** or **rmso**
padding_baud_rate	pb	pb	Lowest baud where padding is needed
virtual_terminal	vt	vt	(Reserved)
width_status_line	wsl	ws	# of columns in status line

Strings:

back_tab	cbt	bt	Back tab
bell	bel	bl	Audible signal (bell)
carriage_return	cr	cr	Carriage return (*)
change_scroll_region	csr	cs	change to lines #1 through #2 (G)
clear_all_tabs	tbc	ct	Clear all tab stops
clear_screen	clear	cl	Clear screen and home cursor (*)
clr_eol	el	ce	Clear to end of line
clr_eos	ed	cd	Clear to end of display (*)
column_address	hpa	ch	Horizontal position absolute (G)
command_character	cmdch	CC	Term. settable cmd char in prototype
cursor_address	cup	cm	Cursor motion to row #1 col #2 (G)
cursor_down	cud1	do	Down one line
cursor_home	home	ho	Home cursor (if no **cup**)
cursor_invisible	civis	vi	Make cursor invisible
cursor_left	cub1	le	Move cursor left one space
cursor_mem_address,	mrcup	CM	Memory relative cursor addressing
cursor_normal	cnorm	ve	Make cursor appear normal (undo **vs/vi**)
cursor_right	cuf1	nd	Non-destructive space (cursor right)
cursor_to_ll	ll	ll	Last line, first column (if no **cup**)
cursor_up	cuu1	up	Upline (cursor up)
cursor_visible	cvvis	vs	Make cursor very visible
delete_character	dch1	dc	Delete character (*)
delete_line	dl1	dl	Delete line (*)
dis_status_line	dsl	ds	Disable status line
down_half_line	hd	hd	Half-line down (forward 1/2 linefeed)
enter_alt_charset_mode	smacs	as	Start alternate character set
enter_blink_mode	blink	mb	Turn on blinking
enter_bold_mode	bold	md	Turn on bold (extra bright) mode
enter_ca_mode	smcup	ti	String to begin programs that use **cup**
enter_delete_mode	smdc	dm	Delete mode (enter)
enter_dim_mode	dim	mh	Turn on half-bright mode
enter_insert_mode	smir	im	Insert mode (enter)
enter_protected_mode	prot	mp	Turn on protected mode
enter_reverse_mode	rev	mr	Turn on reverse video mode
enter_secure_mode	invis	mk	Turn on blank mode (chars invisible)
enter_standout_mode	smso	so	Begin standout mode
enter_underline_mode	smul	us	Start underscore mode
erase_chars	ech	ec	Erase #1 characters (G)
exit_alt_charset_mode	rmacs	ae	End alternate character set
exit_attribute_mode	sgr0	me	Turn off all attributes
exit_ca_mode	rmcup	te	String to end programs that use **cup**
exit_delete_mode	rmdc	ed	End delete mode
exit_insert_mode	rmir	ei	End insert mode
exit_standout_mode	rmso	se	End standout mode
exit_underline_mode	rmul	ue	End underscore mode
flash_screen	flash	vb	Visible bell (may not move cursor)
form_feed	ff	ff	Hardcopy terminal page eject (*)

from_status_line	fsl	fs	Return from status line
init_1string	is1	i1	Terminal initialization string
init_2string	is2	is	Terminal initialization string
init_3string	is3	i3	Terminal initialization string
init_file	if	if	Name of file containing **is**
init_prog	iprog	iP	Path name of program for init
insert_character	ich1	ic	Insert character
insert_line	il1	al	Add new blank line (*)
insert_padding	ip	ip	Insert pad after character inserted(*)
key_a1	ka1	K1	Upper left of keypad
key_a3	ka3	K3	Upper right of keypad
key_b2	kb2	K2	Center of keypad
key_c1	kc1	K4	Lower left of keypad
key_c3	kc3	K5	Lower right of keypad
key_backspace	kbs	kb	Sent by backspace key
key_catab	ktbc	k;	Sent by clear-all-tabs key
key_clear	kclr	kC	Sent by clear screen or erase key
key_ctab	kctab	kt	Sent by clear-tab key
key_dc	kdch1	kD	Sent by delete character key
key_dl	kdl1	kL	Sent by delete line key
key_down	kcud1	kd	Sent by terminal down arrow key
key_eic	krmir	kM	Sent by **rmir** or **smir** in insert mode
key_eol	kel	kE	Sent by clear-to-end-of-line key
key_eos	ked	kS	Sent by clear-to-end-of-screen key
key_f0	kf0	k0	Sent by function key f0
key_f1	kf1	k1	Sent by function key f1
key_f2	kf2	k2	Sent by function key f2
key_f3	kf3	k3	Sent by function key f3
key_f4	kf4	k4	Sent by function key f4
key_f5	kf5	k5	Sent by function key f5
key_f6	kf6	k6	Sent by function key f6
key_f7	kf7	k7	Sent by function key f7
key_f8	kf8	k8	Sent by function key f8
key_f9	kf9	k9	Sent by function key f9
key_f10	kf10	ka	Sent by function key f10
key_home	khome	kh	Sent by home key
key_ic	kich1	kI	Sent by ins char/enter ins mode key
key_il	kil1	kA	Sent by insert line
key_left	kcub1	kl	Sent by terminal left arrow key
key_ll	kll	kH	Sent by home-down key
key_npage	knp	kN	Sent by next-page key
key_ppage	kpp	kP	Sent by previous-page key
key_right	kcuf1	kr	Sent by terminal right arrow key
key_sf	kind	kF	Sent by scroll-forward/down key
key_sr	kri	kR	Sent by scroll-backward/up key
key_stab	khts	kT	Sent by set-tab key
key_up	kcuu1	ku	Sent by terminal up arrow key
keypad_local	rmkx	ke	Out of "keypad transmit" mode
keypad_xmit	smkx	ks	Put terminal in "keypad transmit" mode

lab_f0	lf0	l0	Labels on function key f0 if not f0
lab_f1	lf1	l1	Labels on function key f1 if not f1
lab_f2	lf2	l2	Labels on function key f2 if not f2
lab_f3	lf3	l3	Labels on function key f3 if not f3
lab_f4	lf4	l4	Labels on function key f4 if not f4
lab_f5	lf5	l5	Labels on function key f5 if not f5
lab_f6	lf6	l6	Labels on function key f6 if not f6
lab_f7	lf7	l7	Labels on function key f7 if not f7
lab_f8	lf8	l8	Labels on function key f8 if not f8
lab_f9	lf9	l9	Labels on function key f9 if not f9
lab_f10	lf10	la	Labels on function key f10 if not f10
meta_off	rmm	mo	(Reserved)
meta_on	smm	mm	(Reserved)
newline	nel	nw	Newline (like **cr** followed by **lf**)
pad_char	pad	pc	Pad character (rather than null)
parm_dch	dch	DC	Delete #1 chars (G*)
parm_delete_line	dl	DL	Delete #1 lines (G*)
parm_down_cursor	cud	DO	Move cursor down #1 lines (G*)
parm_ich	ich	IC	Insert #1 blank chars (G*)
parm_index	indn	SF	Scroll forward #1 lines (G)
parm_insert_line	il	AL	Add #1 new blank lines (G*)
parm_left_cursor	cub	LE	Move cursor left #1 spaces (G)
parm_right_cursor	cuf	RI	Move cursor right #1 spaces (G*)
parm_rindex	rin	SR	Scroll backward #1 lines (G)
parm_up_cursor	cuu	UP	Move cursor up #1 lines (G*)
pkey_key	pfkey	pk	Prog funct key #1 to type string #2
pkey_local	pfloc	pl	Prog funct key #1 to execute string #2
pkey_xmit	pfx	px	Prog funct key #1 to xmit string #2
print_screen	mc0	ps	Print contents of the screen
prtr_non	mc5p	pO	Turn on the printer for #1 bytes
prtr_off	mc4	pf	Turn off the printer
prtr_on	mc5	po	Turn on the printer
repeat_char	rep	rp	Repeat char #1 #2 times (G*)
reset_1string	rs1	r1	Reset terminal completely to sane modes
reset_2string	rs2	r2	Reset terminal completely to sane modes
reset_3string	rs3	r3	Reset terminal completely to sane modes
reset_file	rf	rf	Name of file containing reset string
restore_cursor	rc	rc	Restore cursor to position of last **sc**
row_address	vpa	cv	Vertical position absolute (G)
save_cursor	sc	sc	Save cursor position
scroll_forward	ind	sf	Scroll text up
scroll_reverse	ri	sr	Scroll text down
set_attributes	sgr	sa	Define the video attributes #1-#9 (G)
set_tab	hts	st	Set a tab in all rows, current column
set_window	wind	wi	Current window: lines #1-#2 cols #3-#4
tab	ht	ta	Tab to next 8 space hardware tab stop
to_status_line	tsl	ts	Go to status line, column #1
underline_char	uc	uc	Underscore one char and move past it
up_half_line	hu	hu	Half-line up (reverse 1/2 linefeed)

A Sample Entry

The following entry, which describes the Concept−100, is among the more complex entries in the *terminfo* file.

```
concept100|c100|concept|c104|c100-4p|concept 100,
    am, bel=^G, blank=\EH, blink=\EC, clear=^L$<2*>, cnorm=\Ew,
    cols#80, cr=^M$<9>, cub1=^H, cud1=^J, cuf1=\E=,
    cup=\Ea%p1%' '%+%c%p2%' '%+%c,
    cuu1=\E;, cvvis=\EW, db, dch1=\E^A$<16*>, dim=\EE, dl1=\E^B$<3*>,
    ed=\E^C$<16*>, el=\E^U$<16>, eo, flash=\Ek$<20>\EK, ht=\t$<8>,
    il1=\E^R$<3*>, in, ind=^J, .ind=^J$<9>, ip=$<16*>,
    is2=\EU\Ef\E7\E5\E8\E1\ENH\EK\E\200\Eo&\200\Eo\47\E,
    kbs=^h, kcub1=\E>, kcud1=\E<, kcuf1=\E=, kcuu1=\E;,
    kf1=\E5, kf2=\E6, kf3=\E7, khome=\E?,
    lines#24, mir, pb#9600, prot=\EI, rep=\Er%p1%c%p2%' '%+%c$<.2*>,
    rev=\ED, rmcup=\Ev    $<6>\Ep\r\n, rmir=\E\200, rmkx=\Ex,
    rmso=\Ed\Ee, rmul=\Eg, rmul=\Eg, sgr0=\EN\200,
    smcup=\EU\Ev  8p\Ep\r, smir=\E^P, smkx=\EX, smso=\EE\ED,
    smul=\EG, tabs, ul, vt#8, xenl,
```

Entries may continue onto multiple lines by placing white space at the beginning of each line except the first. Lines beginning with "#" are taken as comment lines. Capabilities in *terminfo* are of three types: boolean capabilities which indicate that the terminal has some particular feature, numeric capabilities giving the size of the terminal or the size of particular features, and string capabilities which give a sequence which can be used to perform particular terminal operations.

Types of Capabilities

All capabilities have names. For instance, the fact that the Concept has *automatic margins* (i.e., an automatic return and linefeed when the end of a line is reached) is indicated by the capability **am**. Hence the description of the Concept includes **am**. Numeric capabilities are followed by the character '#' and then the value. Thus **cols**, which indicates the number of columns the terminal has, gives the value '80' for the Concept.

Finally, string valued capabilities, such as **el** (clear to end of line sequence) are given by the two- to five-character capname, an '=', and then a string ending at the next following ','. A delay in milliseconds may appear anywhere in such a capability, enclosed in $<..> brackets, as in **el**=\EK$<3>, and padding characters are supplied by `tputs()` [see CURSES(TI_LIB)] to provide this delay. The delay can be either a number, e.g., '20', or a number followed by an '*', i.e., '3*'. A '*' indicates that the padding required is proportional to the number of lines affected by the operation, and the amount given is the per-affected-unit padding required. (In the case of insert character, the factor is still the number of *lines* affected. This is always one unless the terminal has **xenl** and the software uses it.) When a '*' is specified, it is sometimes useful to give a delay of the form '3.5' to specify a delay per unit

to tenths of milliseconds. (Only one decimal place is allowed.) If the terminal has **xon** defined, the padding information is advisory and will only be used for cost estimates or when the terminal is in raw mode.

A number of escape sequences are provided in the string valued capabilities for easy encoding of characters there. Both **\E** and **\e** map to an ESCAPE character, **^x** maps to a control-x for any appropriate x, and the sequences **\n**, **\l**, **\r**, **\t**, **\b**, **\f**, and **\s** give a newline, linefeed, return, tab, backspace, formfeed, and space. Other escapes include **\^** for **^**, **** for ****, **\,** for comma, **\:** for :, and **\0** for null. (**\0** will produce **\200**, which does not terminate a string but behaves as a null character on most terminals.) Finally, characters may be given as three octal digits after a ****.

Sometimes individual capabilities must be commented out. To do this, put a period before the capability name. For example, see the second **ind** in the example above. Note that capabilities are defined in a left-to-right order and therefore, a prior definition will override a later definition.

Basic Capabilities

The number of columns on each line for the terminal is given by the **cols** numeric capability. If the terminal is a CRT, then the number of lines on the screen is given by the **lines** capability. If the terminal wraps around to the beginning of the next line when it reaches the right margin, then it should have the **am** capability. If the terminal can clear its screen, leaving the cursor in the home position, then this is given by the **clear** capability. If the terminal overstrikes (rather than clearing a position when a character is struck over) then it should have the **os** capability. If the terminal is a printing terminal, with no soft copy unit, give it both **hc** and **os.** (**os** applies to storage scope terminals, such as Tektronix 4010 series, as well as hard copy and APL terminals.) If there is a code to move the cursor to the left edge of the current row, give this as **cr.** (Normally this will be carriage return, control M.) If there is a code to produce an audible signal (bell, beep, etc) give this as **bel.** If the terminal uses the xon-xoff flow-control protocol, like most terminals, specify **xon.**

If there is a code to move the cursor one position to the left (such as backspace) that capability should be given as **cub1.** Similarly, codes to move to the right, up, and down should be given as **cuf1**, **cuu1**, and **cud1**. These local cursor motions should not alter the text they pass over; for example, '**cuf1=**\s' would not normally be used, because the space would erase the character moved over.

A very important point here is that the local cursor motions encoded in *terminfo* are undefined at the left and top edges of a CRT terminal. Programs should never attempt to backspace around the left edge, unless **bw** is given, and never attempt to go up locally off the top. In order to scroll text up, a program will go to the bottom left corner of the screen and send the **ind** (index) string.

To scroll text down, a program goes to the top left corner of the screen and sends the **ri** (reverse index) string. The strings **ind** and **ri** are undefined when not on their respective corners of the screen.

Parameterized versions of the scrolling sequences are **indn** and **rin** which have the same semantics as **ind** and **ri** except that they take one parameter, and scroll that many lines. They are also undefined except at the appropriate edge of the screen.

The **am** capability tells whether the cursor sticks at the right edge of the screen when text is output, but this does not necessarily apply to a **cuf1** from the last column. The only local motion which is defined from the left edge is if **bw** is given, then a **cub1** from the left edge will move to the right edge of the previous row. If **bw** is not given, the effect is undefined. This is useful for drawing a box around the edge of the screen, for example. If the terminal has switch selectable automatic margins, the *terminfo* file usually assumes that this is on; i.e., **am**. If the terminal has a command which moves to the first column of the next line, that command can be given as **nel** (newline). It does not matter if the command clears the remainder of the current line, so if the terminal has no **cr** and **lf** it may still be possible to craft a working **nel** out of one or both of them.

These capabilities suffice to describe hardcopy and "glass-tty" terminals. Thus the model 33 Teletype is described as

```
33|tty33|tty|model 33 teletype,
    bel=^G, cols#72, cr=^M, cud1=^J, hc,
    ind=^J, os,
```

while the Lear Siegler ADM−3 is described as

```
adm3|lsi adm3,
    am, bel=^G, clear=^Z, cols#80, cr=^M,
    cub1=^H, cud1=^J, ind=^J, lines#24,
```

Parameterized Strings

Cursor addressing and other strings requiring parameters in the terminal are described by a parameterized string capability. For example, to address the cursor, the **cup** capability is given, using two parameters: the row and column to address to. (Rows and columns are numbered from zero and refer to the physical screen visible to the user, not to any unseen memory.) If the terminal has memory relative cursor addressing, that can be indicated by **mrcup.**

The parameter mechanism uses a stack and special % codes to manipulate it in the manner of a Reverse Polish Notation calculator. Typically a sequence will push one of the parameters onto the stack and then print it in some format. Often more complex operations are necessary. Binary operations are in postfix form with the operands in the usual order. That is, to get x-5 one would use "%gx%{5}%-".

The % encodings have the following meanings:

%%	outputs '%'
%d	print pop() as a decimal number
%3d	print pop() in a field at least 3 spaces wide
%03d	use leading zeros to fill
%c	print pop() as a character
%s	print pop() as a character string
%p[1-9]	push *i*th parm
%P[a-z]	set variable [a-z] to pop()
%g[a-z]	get variable [a-z] and push it
%'c'	push char constant c
%{nn}	push integer constant nn
%l	push strlen(pop())
%+ %-	
%* %/ %m	arithmetic (%m is mod): push(pop() op pop())
%& %| %^	bit operations: push(pop() op pop())
%= %> %<	logical operations: push(pop() op pop())
%! %	unary operations push(op pop())
%i	add 1 to first two parms (for ANSI terminals)
%? expr %t thenpart %e elsepart %;	if-then-else, %e elsepart is optional. else-if's are possible: %? c_1 %t b_1 %e c_2 %t b_2 %e c_3 %t b_3 %e b_4; (c*i* are conditions, b*i* are bodies)

Consider the Hewlett-Packard 2645, which, to get to row 3 and column 12, needs to be sent \E&a12c03Y padded for 6 milliseconds. Note that the order of the rows and columns is inverted here, and that the row and column are printed as two digits. Thus its **cup** capability is "**cup**=\E&a%p2%2dc%p1%2dY$<6>".

The Micro-Term ACT-IV needs the current row and column sent preceded by a ^T, with the row and column simply encoded in binary, "**cup**=^T%p1%c%p2%c". Terminals which use "%c" need to be able to backspace the cursor (**cub1**), and to move the cursor up one line on the screen (**cuu1**). This is necessary because it is not always safe to transmit **\n**, **^D**, and **\r**, as the system may change or discard them. (The library routines dealing with *terminfo* set tty modes so that tabs are never expanded, so **\t** is safe to send. This turns out to be essential for the Ann Arbor 4080.)

A final example is the LSI ADM-3a, which uses row and column offset by a blank character, thus "**cup**=\E=%p1%'\s'%+%c%p2%'\s'%+%c". After sending '\E=', this pushes the first parameter, pushes the ASCII value for a space (32), adds them (pushing the sum on the stack in place of the two previous values) and outputs that value as a character. Then the same is done for the second parameter. More complex arithmetic is possible using the stack.

Cursor Motions

If the terminal has a fast way to home the cursor (to very upper left corner of screen) then this can be given as **home**; similarly a fast way of getting to the lower left-hand corner can be given as **ll**; this may involve going up with **cuu1** from the home position, but a program should never do this itself (unless **ll** does) because it can make no assumption about the effect of moving up from the home position. Note that the home position is the same as addressing to (0,0): to the top left corner of the screen, not of memory. (Thus, the **\EH** sequence on Hewlett-Packard terminals cannot be used for **home** without losing some of the other features of the terminal.)

If the terminal has row or column absolute cursor addressing, these can be given as single parameter capabilities **hpa** (horizontal position absolute) and **vpa** (vertical position absolute). Sometimes these are shorter than the more general two parameter sequence (as with the HP2645) and can be used in preference to **cup.** If there are parameterized local motions (e.g., move *n* spaces to the right) these can be given as **cud, cub, cuf,** and **cuu** with a single parameter indicating how many spaces to move. These are primarily useful if the terminal does not have **cup,** such as the Tektronix 4025.

Area Clears

If the terminal can clear from the current position to the end of the line, leaving the cursor where it is, this should be given as **el.** If the terminal can clear from the current position to the end of the display, then this should be given as **ed.** The **ed** capability is only defined from the first column of a line. (Thus, it can be simulated by a request to delete a large number of lines, if a true **ed** is not available.)

Insert/Delete Line

If the terminal can open a new blank line before the line where the cursor is, this should be given as **il1**; this is done only from the first position of a line. The cursor must then appear on the newly blank line. If the terminal can delete the line which the cursor is on, then this should be given as **dl1**; this is done only from the first position on the line to be deleted. Versions of **il1** and **dl1** which take a single parameter and insert or delete that many lines can be given as **il** and **dl.** If the terminal has a settable destructive scrolling region (like the VT100) the command to set this can be described with the **csr** capability, which takes two parameters: the top and bottom lines of the scrolling region. The cursor position is, alas, undefined after using this command. It is possible to get the effect of insert or delete line using this command — the **sc** and **rc** (save and restore cursor) commands are also useful. Inserting lines at the top or bottom of the screen can also be done using **ri** or **ind** on many terminals without a true insert/delete line, and is often faster even on terminals with those features.

To determine whether a terminal has destructive scrolling regions or non-destructive scrolling regions, create a scrolling region in the middle of the screen, place data on the bottom line of the scrolling region, move the cursor

to the top line of the scrolling region, and do a reverse index **ri** followed by a delete line **dl1** or index **ind**. If the data that was originally on the bottom line of the scrolling region was restored into the scrolling region by the **dl1** or **ind**, then the terminal has non-destructive scrolling regions. Otherwise, it has destructive scrolling regions.

If the terminal has the ability to define a window as part of memory, which all commands affect, it should be given as the parameterized string **wind.** The four parameters are the starting and ending lines in memory and the starting and ending columns in memory, in that order.

If the terminal can retain display memory above, then the **da** capability should be given; if display memory can be retained below, then **db** should be given. These indicate that deleting a line or scrolling a full screen may bring non-blank lines up from below or that scrolling back with **ri** may bring down non-blank lines.

Insert/Delete Character

There are two basic kinds of intelligent terminals with respect to insert/delete character which can be described using *terminfo.* The most common insert/delete character operations affect only the characters on the current line and shift characters off the end of the line rigidly. Other terminals, such as the Concept 100 and the Perkin Elmer Owl, make a distinction between typed and untyped blanks on the screen, shifting upon an insert or delete only to an untyped blank on the screen which is either eliminated, or expanded to two untyped blanks. Users can determine the kind of terminal they have by clearing the screen and then typing text separated by cursor motions. Type "abc def" using local cursor motions (not spaces) between the "abc" and the "def". Then position the cursor before the "abc" and put the terminal in insert mode. If typing characters causes the rest of the line to shift rigidly and characters to fall off the end, then the terminal does not distinguish between blanks and untyped positions. If the "abc" shifts over to the "def" which then move together around the end of the current line and onto the next during the insert, then this is the second type of terminal, and the description should be given the capability **in**, which stands for "insert null. While these are two logically separate attributes (one line vs. multiline insert mode, and special treatment of untyped spaces) we have seen no terminals whose insert mode cannot be described with the single attribute.

The *terminfo* database can describe both terminals which have an insert mode, and terminals which send a simple sequence to open a blank position on the current line. Give as **smir** the sequence to get into insert mode. Give as **rmir** the sequence to leave insert mode. Now give as **ich1** any sequence needed to be sent just before sending the character to be inserted. Most terminals with a true insert mode will not give **ich1**; terminals which send a sequence to open a screen position should give it here. (If the terminal has both, insert mode is usually preferable to **ich1**. Both should not be given unless the terminal actually requires both to be used in combination.) If post insert padding is needed, give this as a number of milliseconds in **ip** (a string

option). Any other sequence which may need to be sent after an insert of a single character may also be given in **ip**. If the terminal needs both to be placed into an 'insert mode' and a special code to precede each inserted character, then both **smir/rmir** and **ich1** can be given, and both will be used. The **ich** capability, with one parameter, *n*, will repeat the effects of **ich1** *n* times.

It is occasionally necessary to move around while in insert mode to delete characters on the same line (e.g., if there is a tab after the insertion position). If the terminal allows motion while in insert mode the capability **mir** cna be given to speed up inserting in this case. Omitting **mir** will affect only speed. Some terminals (notably Datamedia's) must not have **mir** because of the way their insert mode works.

Finally, **dch1** can be specified to delete a single character, **dch** with one parameter, *n*, to delete *n* characters and delete mode by giving **smdc** and **rmdc** to enter and exit delete mode (any mode the terminal needs to be placed in for **dch1** to work).

A command to erase *n* characters (equivalent to outputting *n* blanks without moving the cursor) can be given as **ech** with one parameter.

Highlighting, Underlining, and Visible Bells

If the terminal has one or more kinds of display attributes, these can be represented in a number of different ways. One display form should be chosen as *standout mode*, representing a good, high contrast, easy-on-the-eyes, format for highlighting error messages and other attention getters. (If there is a choice, reverse video plus half-bright is good, or reverse video alone.) The sequences to enter and exit standout mode are given as **smso** and **rmso**, respectively. If the code to change into or out of standout mode leaves one or even two blank spaces on the screen, as the TeleVideo 912 and Teleray 1061 do, then **xmc** should be given to tell how many spaces are left.

Codes to begin underlining and end underlining can be given as **smul** and **rmul**, respectively. If the terminal has a code to underline the current character and move the cursor one space to the right, such as the Micro-Term MIME, this can be given as **uc**.

Other capabilities to enter various highlighting modes include **blink** (blinking), **bold** (bold or extra bright), **dim** (dim or half-bright), **invis** (blanking or invisible text), **prot** (protected), **rev** (reverse video), **sgr0** (turn off *all* attribute modes), **smacs** (enter alternate character set mode), and **rmacs** (exit alternate character set mode). Turning on any of these modes singly may or may not turn off other modes.

If there is a sequence to set arbitrary combinations of modes, this should be given as **sgr** (set attributes), taking nine parameters. Each parameter is either 0 or non-zero, as the corresponding attribute is on or off. The nine parameters are, in order: standout, underline, reverse, blink, dim, bold, blank, protect, and alternate character set. Not all modes need be supported by **sgr**, only those for which corresponding separate attribute commands exist.

Terminals with the "magic cookie" glitch (**xmc**) deposit special "cookies" when they receive mode-setting sequences, which affect the display algorithm rather than having extra bits for each character. Some terminals, such as the Hewlett-Packard 2621, automatically leave standout mode when they move to a new line or the cursor is addressed. Programs using standout mode should exit standout mode before moving the cursor or sending a newline, unless the **msgr** capability, asserting that it is safe to move in standout mode, is present.

If the terminal has a way of flashing the screen to indicate an error quietly (a bell replacement) then this can be given as **flash**; it must not move the cursor.

If the cursor needs to be made more visible than normal when it is not on the bottom line (to make, for example, a non-blinking underline into an easier to find block or blinking underline) give this sequence as **cvvis.** If there is a way to make the cursor completely invisible, give that as **civis.** The capability **cnorm** should be given which undoes the effects of both of these modes.

If the terminal needs to be in a special mode when running a program that uses these capabilities, the codes to enter and exit this mode can be given as **smcup** and **rmcup.** This arises, for example, from terminals like the Concept with more than one page of memory. If the terminal has only memory relative cursor addressing and not screen relative cursor addressing, a one screen-sized window must be fixed into the terminal for cursor addressing to work properly. This is also used for the Tektronix 4025, where **smcup** sets the command character to be the one used by *terminfo.*

If the terminal correctly generates underlined characters (with no special codes needed) even though it does not overstrike, then the capability **ul** should be given. If overstrikes are erasable with a blank, then this should be indicated by giving **eo.**

Keypad

If the terminal has a keypad that transmits codes when the keys are pressed, this information can be given. Note that it is not possible to handle terminals where the keypad only works in local (this applies, for example, to the unshifted Hewlett-Packard 2621 keys). If the keypad can be set to transmit or not transmit, give these codes as **smkx** and **rmkx.** Otherwise the keypad is assumed to always transmit.

The codes sent by the left arrow, right arrow, up arrow, down arrow, and home keys can be given as **kcub1, kcuf1, kcuu1, kcud1,** and **khome**, respectively. If there are function keys such as f0, f1, ..., f10, the codes they send can be given as **kf0, kf1, ..., kf10.** If these keys have labels other than the default f0 through f10, the labels can be given as **lf0, lf1, ..., lf10.** The codes transmitted by certain other special keys can be given: **kll** (home down), **kbs** (backspace), **ktbc** (clear all tabs), **kctab** (clear the tab stop in this column), **kclr** (clear screen or erase key), **kdch1** (delete character), **kdl1** (delete line), **krmir** (exit insert mode), **kel** (clear to end of line), **ked** (clear to end of screen), **kich1** (insert character or enter insert mode), **kil1** (insert line), **knp** (next page), **kpp** (previous page), **kind** (scroll forward/down), **kri** (scroll

backward/up), **khts** (set a tab stop in this column). In addition, if the keypad has a 3 by 3 array of keys including the four arrow keys, the other five keys can be given as **ka1, ka3, kb2, kc1,** and **kc3.** These keys are useful when the effects of a 3 by 3 directional pad are needed.

Strings to program function keys can be given as **pfkey, pfloc,** and **pfx.** Each of these strings takes two parameters: the function key number to program (from 0 to 10) and the string to program it with. Function key numbers out of this range may program undefined keys in a terminal dependent manner. The difference between the capabilities is that **pfkey** causes pressing the given key to be the same as the user typing the given string; **pfloc** causes the string to be executed by the terminal in local; and **pfx** causes the string to be transmitted to the computer.

Tabs and Initialization

If the terminal has hardware tabs, the command to advance to the next tab stop can be given as **ht** (usually control I). A "backtab" command which moves leftward to the next tab stop can be given as **cbt.** By convention, if the terminal modes indicate that tabs are being expanded by the computer rather than being sent to the terminal, programs should not use **ht** or **cbt** even if they are present, since the user may not have the tab stops properly set. If the terminal has hardware tabs which are initially set every *n* spaces when the terminal is powered up, the numeric parameter **it** is given, showing the number of spaces the tabs are set to. This is normally used to determine whether to set the mode for hardware tab expansion, and whether to set the tab stops. If the terminal has tab stops that can be saved in nonvolatile memory, the *terminfo* description can assume that they are properly set. If there are commands to set and clear tab stops, they can be given as **tbc** (clear all tab stops) and **hts** (set a tab stop in the current column of every row).

Other capabilities include **is1, is2,** and **is3,** initialization strings for the terminal, **iprog,** the path name of a program to be run to initialize the terminal, and **if,** the name of a file containing long initialization strings. These strings are expected to set the terminal into modes consistent with the rest of the *terminfo* description. They must be sent to the terminal each time the user logs in and be output in the following order: run the program **iprog,** output **is1; is2;** set tabs using **tbc** and **hts;** print the file **if;** and finally output **is3.** See the **USAGE** section below. Most initialization is done with **is2.** Special terminal modes can be set up without duplicating strings by putting the common sequences in **is2** and special cases in **is1** and **is3.** A pair of sequences that does a harder reset from a totally unknown state can be analogously given as **rs1, rs2, rf,** and **rs3,** analogous to **is*** and **if.** These strings should be output when the terminal gets into a wedged state. Commands are normally placed in **rs*** and **rf** only if they produce annoying effects on the screen and are not necessary when logging in. For example, the command to set a terminal into 80-column mode would normally be part of **is2,** but on some terminals it causes an annoying glitch of the screen and is not normally needed since the terminal is usually already in 80 column mode. Therefore, the command is

usually placed in **rs1**, not **is2**, for those terminals.

If a more complex sequence is needed to set the tabs than can be described by using **tbc** and **hts**, the sequence can be placed in **is2** or **if.**

Delays

Certain capabilities control padding in the tty driver. These are primarily needed by hard copy terminals. Delays embedded in the capabilities **cr, ind, cub1, ff,** and **tab** can be used to set the appropriate delay bits in the tty driver. If **pb** (padding baud rate) is given, these values can be ignored at baud rates below the value of **pb.**

Status Line

If the terminal has an extra "status line" that is not normally used by software, this fact can be indicated. If the status line is viewed as an extra line below the bottom line, into which one can cursor address normally (such as the Heathkit h19's 25th line, or the 24th line of a VT100 which is set to a 23-line scrolling region), the capability **hs** should be given. Special strings to go to the beginning of the status line and to return from the status line can be given as **tsl** and **fsl.** (**fsl** must leave the cursor position in the same place it was before **tsl.** If necessary, the **sc** and **rc** strings can be included in **tsl** and **fsl** to get this effect.) The parameter **tsl** takes one parameter, which is the column number of the status line the cursor is to be moved to. If escape sequences and other special commands, such as **tab** and **el**, work while in the status line, the flag **eslok** can be given. A string which turns off the status line (or otherwise erases its contents) should be given as **dsl.** If the terminal has commands to save and restore the position of the cursor, give them as **sc** and **rc.** The status line is normally assumed to be the same width as the rest of the screen, e.g., **cols.** If the status line is a different width (possibly because the terminal does not allow an entire line to be loaded) the width, in columns, can be indicated with the numeric parameter **wsl.**

Miscellaneous

If the terminal requires other than a null (zero) character as a pad, then this can be given as **pad.** Only the first character of the **pad** string is used.

If the terminal can move up or down half a line, this can be indicated with **hu** (half-line up) and **hd** (half-line down). This is primarily useful for superscripts and subscripts on hardcopy terminals. If a hardcopy terminal can eject to the next page (form feed), give this as **ff** (usually control L).

If there is a command to repeat a given character a given number of times (to save time transmitting a large number of identical characters) this can be indicated with the parameterized string **rep.** The first parameter is the character to be repeated and the second is the number of times to repeat it. Thus, `tparm(repeat_char, 'x', 10)` is the same as 'xxxxxxxxxx'.

If the terminal has a settable command character, such as the Tektronix 4025, this can be indicated with **cmdch.** A prototype command character is chosen which is used in all capabilities. This character is given in the **cmdch** capability to identify it.

Terminal descriptions that do not represent a specific kind of known terminal, such as **switch, dialup, patch,** and **network,** should include the **gn** (generic) capability so that programs can complain that they do not know how to talk to the terminal.

If the terminal uses xon/xoff handshaking for flow control, give **xon.** Padding information should still be included so that routines can make better decisions about costs, but actual pad characters will not be transmitted.

If the terminal has more lines of memory than will fit on the screen at once, the number of lines of memory can be indicated with **lm.** A value of **lm#0** indicates that the number of lines is not fixed, but that there is still more memory than fits on the screen.

Media copy strings which control an auxiliary printer connected to the terminal can be given as **mc0**: print the contents of the screen, **mc4**: turn off the printer, and **mc5**: turn on the printer. When the printer is on, all text sent to the terminal will be sent to the printer. It is undefined whether the text is also displayed on the terminal screen when the printer is on. A variation **mc5p** takes one parameter, and leaves the printer on for as many characters as the value of the parameter, then turns the printer off. The parameter should not exceed 255. All text, including **mc4,** is transparently passed to the printer while an **mc5p** is in effect.

Special Cases

The working model used by *terminfo* fits most terminals reasonably well. However, some terminals do not completely match that model, requiring special support by *terminfo*. These are not meant to be construed as deficiencies in the terminals; they are just differences between the working model and the actual hardware.

Terminals which can not display `tilde` characters, such as certain Hazeltine terminals, should indicate **hz**.

Terminals which ignore a linefeed immediately after an **am** wrap, such as the Concept 100, should indicate **xenl**. Those terminals whose cursor remains on the right-most column until another character has been received, rather than wrapping immediately upon receiving the right-most character, such as the VT100, should also indicate **xenl**.

If **el** is required to get rid of standout (instead of writing normal text on top of it), **xhp** should be given.

Those Teleray terminals whose tabs turn all characters moved over to blanks, should indicate **xt**. This capability is also taken to mean that it is not possible to position the cursor on top of a "magic cookie"; therefore, to erase standout mode it is instead necessary to use delete and insert line.

The Beehive Superbee terminals which do not transmit the escape or control C characters, should specify **xsb**, indicating that the f1 key is to used for escape and f2 for control C.

Similar Terminals

If there are two very similar terminals, one can be defined as being just like the other with certain exceptions. The string capability **use** can be given with the name of the similar terminal. The capabilities given before **use** override those in the terminal type invoked by **use**. A capability can be cancelled by placing **xx@** to the left of the capability definition. For example, the entry

```
tty4424-2|Teletype 4424 - displ func grp ii
    rev@, sgr@, smul@, use=tty4424,
```

defines a Teletype 4424 terminal that does not have the **rev**, **sgr**, and **smul** capabilities, and hence cannot do highlighting. This is useful for different modes for a terminal, or for different user preferences. More than one **use** capability may be given.

FILES

`/usr/lib/terminfo/?/*` Compiled terminal description database

SEE ALSO

CURSES(TI_LIB), PRINTF(BA_LIB), TIC(TI_CMD).

USAGE

Administrator and Application Program.

As described in the **Tabs and Initialization** section above, a terminal's initialization strings, **is1**, **is2**, and **is3**, if defined, must be output before a *curses* program is run. An available mechanism for outputting such strings is the TPUT(TI_CMD) command. A sample program which performs such initialization follows:

```
eval 'tput iprog'
tput is1
tput is2
if [ -n "'tput tab'" ]
then
stty tabs
else
stty -tabs
fi
tabs
cat -s "'tput if'"
tput is3
echo "\r\c"
```

A terminal's reset strings, **rs1**, **rs2**, and **rs3**, may be output in a similar manner.

The most effective way to prepare a terminal description is by imitating the description of a similar terminal in *terminfo* and to build up a description gradually, using partial descriptions with VI(AU_CMD) to check that they are correct. To easily test a new terminal description the environment variable `TERMINFO` can be set to the pathname of a directory containing the compiled description and programs will look there rather than in `/usr/lib/terminfo`. To get the padding for insert line right a severe test is to comment out **xon**, edit a copy of a large file at 9600 baud with VI(AU_CMD), delete 16 or so lines from the middle of the screen, then hit the 'u' key several times quickly. If the terminal messes up, more padding is usually needed. A similar test can be used for insert character.

FUTURE DIRECTIONS

The ability to output a terminal's initialization strings or reset strings through the specification of a single argument will be incorporated into TPUT(TI_CMD).

The capabilities of *terminfo* will be enhanced to include a way to specify mandatory padding; to define additional function keys; to support additional % codes within the string capabilities; and to support line drawing alternate character sets.

LEVEL

Level 1.

Chapter 14
Library Routines

NAME

curses — CRT screen handling and optimization package

SYNOPSIS

```
#include <curses.h>
```

DESCRIPTION

The *curses* library routines give the user a method of updating screens with reasonable optimization. A program using these routines must be compiled with the `-lcurses` option of `cc`.

In order to initialize the routines, the routine `initscr()` must be called before any of the other routines that deal with windows and screens are used. The routine `endwin()` should be called before exiting. To get character-at-a-time input without echoing (most interactive, screen oriented-programs want this), the following sequence should be used:

```
initscr(); cbreak(); noecho();
```

Most programs would additionally use the sequence:

```
nonl(); intrflush(stdscr, FALSE);
keypad(stdscr, TRUE);
```

Before a *curses* program is run, a terminal's tabs stops should be set and its initialization strings, if defined, must be output. See TERMINFO(TI_ENV) for further details.

The *curses* library permits manipulation of data structures called *windows* which can be thought of as two-dimensional arrays of characters representing all or part of a CRT screen. A default window called `stdscr` is supplied, and others can be created with `newwin()`. Windows are referred to by variables declared as "**WINDOW ***". These data structures are manipulated with routines described below, among which the most basic are `move()` and `addch()`. (More general versions of these routines are included with names beginning with **w**, allowing one to specify a window. The routines not beginning with **w** affect `stdscr`.) Then `refresh()` is called, telling the routines to make the user's CRT screen look like `stdscr`. The characters in a window are actually of type **chtype**, so that other information about the character may also be stored with each character.

Special windows called *pads* may also be manipulated. These are windows which are not constrained to the size of the screen and whose contents need not be completely displayed.

In addition to drawing characters on the screen, video attributes may be included which cause the characters to show up in such modes as underlined or in reverse video on terminals that support such display enhancements. On input, *curses* is also able to translate arrow and function keys that transmit escape sequences into single values. The video attributes and input values use names such as **A_REVERSE** and **KEY_LEFT**.

The environment variables `LINES` and `COLUMNS` may also be set to override *terminfo*'s idea of how large a screen is. These may be used in a

Teletype 5620 layer, for example, where the size of a screen is changeable.

If the environment variable TERMINFO is defined, any program using *curses* will check for a local terminal definition before checking in the standard place. For example, if TERM is set to ``tty4424'', then the compiled terminal definition is found in /usr/lib/terminfo/t/tty4424. (The "t" is copied from the first letter of "tty4424" to avoid creation of huge directories.) However, if TERMINFO is set to $HOME/myterms, *curses* will first check $HOME/myterms/t/tty4424, and if that fails, will then check /usr/lib/terminfo/t/tty4424. This is useful for developing experimental definitions or when write permission in /usr/lib/terminfo is not available.

The integer variables LINES and COLS are defined in <curses.h> and will be filled in by initscr() with the size of the screen. The constants TRUE and FALSE have the values 1 and 0, respectively.

The *curses* routines also define the WINDOW * variable curscr which is used for certain low-level operations like clearing and redrawing a garbaged screen. curscr can be used in only a few routines. If the window argument to clearok() is curscr, the next call to wrefresh() with any window will cause the screen to be cleared and repainted from scratch. If the window argument to wrefresh() is curscr, the screen in immediately cleared and repainted from scratch. This is how most programs would implement a "repaint-screen" function.

Routines

Many of the following routines have two or more versions. The routines prefixed with **w** require a window argument. The routines prefixed with **p** require a pad argument. Those without a prefix generally use stdscr.

The routines prefixed with **mv** require x and y coordinates to move to before performing the appropriate action. The **mv** routines imply a call to move before the call to the other routine. The upper left corner is always (0,0), not (1,1).

The routines prefixed with **mvw** take both a window argument and x and y coordinates. The window argument is always specified before the coordinates.

In each case, win is the window affected and pad is the pad affected; win and pad are always of type WINDOW. Option setting routines require a boolean flag bf with the value TRUE or FALSE; bf is always of type bool. The variables ch and attrs below are always of type chtype. The types WINDOW, bool, and chtype are defined in <curses.h>. All other arguments are integers.

See the **RETURN VALUE** section for information on the values returned by the routines described below.

Overall Screen Manipulation

`initscr()`

The first routine called should almost always be `initscr()`. This will determine the terminal type and initialize all *curses* data structures. `initscr()` also arranges that the first call to `refresh()` will clear the screen. If errors occur, `initscr()` will write an appropriate error message to standard error and exit. If the program wants a indication of error conditions, `newterm()` should be used instead of `initscr()`.

`endwin()`

A program should always call `endwin()` before exiting or escaping from *curses* mode temporarily. This routine will restore tty modes, move the cursor to the lower left corner of the screen and reset the terminal into the proper non-visual mode. To resume after a temporary escape, call `refresh()` or `doupdate()`.

```
SCREEN *newterm(type, outfd, infd)
char *type;
FILE *outfd, *infd;
```

A program which outputs to more than one terminal should use `newterm()` for each terminal instead of `initscr()`. A program which wants an indication of error conditions, so that it may continue to run in a line-oriented mode if the terminal cannot support a screen-oriented program, would also use this routine. The routine `newterm()` should be called once for each terminal. It returns a variable of type `SCREEN *` which should be saved as a reference to that terminal. The arguments are the type of the terminal to be used in place of `TERM`, a file pointer for output to the terminal, and another file pointer for input from the terminal. The program must also call `endwin()` for each terminal being used when it is done running.

```
SCREEN *set_term(new)
SCREEN *new;
```

This routine is used to switch between different terminals. The screen reference `new` becomes the new current terminal. The previous terminal is returned by the routine. This is the only routine which manipulates `SCREEN` pointers; all other routines affect only the current terminal.

Window and Pad Manipulation

`refresh()`

```
wrefresh(win)
WINDOW *win;
```

These routines must be called to get any output on the terminal, as other routines merely manipulate data structures. `wrefresh()` copies the named window to the physical terminal screen, taking into

account what is already there in order to do optimizations. `refresh()` is the same, using `stdscr` as a default screen. Unless `leaveok()` has been enabled, the physical cursor of the terminal is left at the location of the window's cursor.

NOTE: `refresh` is a macro.

```
wnoutrefresh(win)
WINDOW *win;
```

```
doupdate( )
```

These two routines allow multiple updates with more efficiency than `wrefresh()` alone. In addition to all of the window structures, *curses* keeps two data structures representing the terminal screen: a **physical** screen, describing what is actually on the screen, and a **virtual** screen, describing what the programmer wants to have on the screen.

The routine `wrefresh()` works by first calling `wnoutrefresh()`, which copies the named window to the virtual screen, and then calling `doupdate()`, which compares the virtual screen to the physical screen and does the actual update. If the programmer wishes to output several windows at once, a series of calls to `wrefresh()` will result in alternating calls to `wnoutrefresh()` and `doupdate()`, causing several bursts of output to the screen. By first calling `wnoutrefresh()` for each window, it is then possible to call `doupdate()` once, resulting in only one burst of output, with probably fewer total characters transmitted and certainly less CPU time used.

```
WINDOW *newwin(nlines, ncols, begin_y, begin_x)
int nlines, ncols, begin_y, begin_x;
```

Create a new window with the given number of lines, `nlines`, and columns, `ncols`. The upper left corner of the window is at line `begin_y`, column `begin_x`. If either `nlines` or `ncols` is zero, they will be defaulted to `LINES - begin_y` and `COLS - begin_x`. A new full-screen window is created by calling `newwin(0,0,0,0)`.

```
mvwin(win, y, x)
WINDOW *win;
int y, x;
```

Move the window so that the upper left corner will be at position (`x`, `y`). If the move would cause the window to be off the screen, it is an error and the window is not moved.

```
WINDOW *subwin(orig, nlines, ncols, begin_y,
begin_x)
WINDOW *orig;
int nlines, ncols, begin_y, begin_x;
```

Create a new window with the given number of lines, `nlines`, and columns, `ncols`. The window is at position (`begin_y`,

begin_x) on the screen. (This position is relative to the screen, and not to the window orig.) The window is made in the middle of the window orig, so that changes made to one window will affect both windows. When using this routine, often it will be necessary to call touchwin() before calling wrefresh().

```
delwin(win)
WINDOW *win;
```

Deletes the named window, freeing up all memory associated with it. In the case of overlapping windows, subwindows should be deleted before the main window.

```
WINDOW *newpad(nlines, ncols)
int nlines, ncols;
```

Creates a new pad data structure. A pad is like a window, except that it is not restricted by the screen size, and is not necessarily associated with a particular part of the screen. Pads can be used when a large window is needed, and only a part of the window will be on the screen at one time. Automatic refreshes of pads (e.g. from scrolling or echoing of input) do not occur. It is not legal to call refresh() with a pad as an argument; the routines prefresh() or pnoutrefresh() should be called instead. Note that these routines require additional parameters to specify the part of the pad to be displayed and the location on the screen to be used for display.

```
prefresh(pad, pminrow, pmincol, sminrow, smincol,
smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow,
smaxcol;
pnoutrefresh(pad, pminrow, pmincol, sminrow,
smincol, smaxrow, smaxcol)
WINDOW *pad;
int pminrow, pmincol, sminrow, smincol, smaxrow,
smaxcol;
```

These routines are analogous to wrefresh() and wnoutrefresh() except that pads, instead of windows, are involved. The additional parameters are needed to indicate what part of the pad and screen are involved. pminrow and pmincol specify the upper left corner, in the pad, of the rectangle to be displayed. sminrow, smincol, smaxrow, and smaxcol specify the edges, on the screen, of the rectangle to be displayed in. The lower right corner in the pad of the rectangle to be displayed is calculated from the screen coordinates, since the rectangles must be the same size. Both rectangles must be entirely contained within their respective structures.

Output

These routines are used to "draw" text on windows.

```
addch(ch)
chtype ch;

waddch(win, ch)
WINDOW *win;
chtype ch;

mvaddch(y, x, ch)
int y, x;
chtype ch;

mvwaddch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;
```

The character `ch` is put into the window at the current cursor position of the window and the position of the window cursor is advanced. Its function is similar to that of `putchar`. At the right margin, an automatic newline is performed. At the bottom of the scrolling region, if `scrollok()` is enabled, the scrolling region will be scrolled up one line.

If `ch` is a tab, newline, or backspace, the cursor will be moved appropriately within the window. A newline also does a `clrtoeol()` before moving. Tabs are considered to be at every eighth column. If `ch` is another control character, it will be drawn in the ^X notation. Calling `winch()` after adding a control character will not return the control character, but instead will return the representation of the control character.

Video attributes can be combined with a character by or-ing them into the parameter. This will result in these attributes also being set. (The intent here is that text, including attributes, can be copied from one place to another using `inch()` and `addch()`.) See `standout()` below.

NOTE: `addch`, `mvaddch`, and `mvwaddch` are macros.

```
addstr(str)
char *str;

waddstr(win, str)
WINDOW *win;
char *str;

mvaddstr(y, x, str)
int y, x;
char *str;

mvwaddstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;
```

These routines write all the characters of the null terminated character string `str` on the given window. It is equivalent to calling `waddch()` once for each character in the string.

NOTE: `addstr`, `mvaddstr`, and `mvwaddstr` are macros.

```
attroff(attrs)
int attrs;
wattroff(win, attrs)
WINDOW *win;
int attrs;
attron(attrs)
int attrs;
wattron(win, attrs)
WINDOW *win;
int attrs;
attrset(attrs)
int attrs;
wattrset(win, attrs)
WINDOW *win;
int attrs;
standend( )
wstandend(win)
WINDOW *win;
standout( )
wstandout(win)
WINDOW *win;
```

These routines manipulate the current attributes of the named window. These attributes can be any combination of `A_STANDOUT`, `A_REVERSE`, `A_BOLD`, `A_DIM`, `A_BLINK`, and `A_UNDERLINE`. These constants are defined in `<curses.h>` and can be combined with the C | (or) operator.

The current attributes of a window are applied to all characters that are written into the window with `waddch()`. Attributes are a property of the character, and move with the character through any scrolling and insert/delete line/character operations. To the extent possible on the particular terminal, they will be displayed as the graphic rendition of characters put on the screen.

`attrset(attrs)` sets the current attributes of the given window to `attrs`. `attroff(attrs)` turns off the named attributes without turning on or off any other attributes. `attron(attrs)` turns on the named attributes without affecting any others. `standout()` is the same as `attron(A_STANDOUT)`. `standend()` is the same as `attrset(0)`, that is, it turns off all attributes.

NOTE: `attroff`, `attron`, and `attrset` are macros.

```
beep( )
flash( )
```

These routines are used to signal the terminal user. `beep()` will sound the audible alarm on the terminal, if possible, and if not, will flash the screen (visible bell), if that is possible. `flash()` will flash the screen, and if that is not possible, will sound the audible signal. If neither signal is possible, nothing will happen. Nearly all terminals have an audible signal (bell or beep), but only some can flash the screen.

```
box(win, vert, hor)
WINDOW *win;
chtype vert, hor;
```

A box is drawn around the edge of the window. `vert` and `hor` are the characters the box is to be drawn with. If `vert` and `hor` are 0, then appropriate default characters will be used.

```
erase( )
werase(win)
WINDOW *win;
```

These routines copy blanks to every position in the window.

NOTE: `erase` is a macro.

```
clear( )
wclear(win)
WINDOW *win;
```

These routines are like `erase()` and `werase()`, but they also call `clearok()`, arranging that the screen will be cleared completely on the next call to `wrefresh()` for that window and repainted from scratch.

NOTE: `clear` is a macro.

```
clrtobot( )
wclrtobot(win)
WINDOW *win;
```

All lines below the cursor in this window are erased. Also, the current line to the right of the cursor, inclusive, is erased.

NOTE: `clrtobot` is a macro.

```
clrtoeol( )
wclrtoeol(win)
WINDOW *win;
```

The current line to the right of the cursor, inclusive, is erased.

NOTE: `clrtoeol` is a macro.

```
delay_output(ms)
int ms;
```

Insert `ms` millisecond pause in output. It is not recommended that

this routine be used extensively since padding characters are used rather than a CPU pause.

```
delch( )
wdelch(win)
WINDOW *win;
mvdelch(y, x)
int y, x;
mvwdelch(win, y, x)
WINDOW *win;
int y, x;
```

The character under the cursor in the window is deleted. All characters to the right on the same line are moved to the left one position and the last character on the line is filled with a blank. The cursor position does not change (after moving to `y, x`, if specified). (This does not imply use of the hardware delete character feature.)

NOTE: `delch`, `mvdelch`, and `mvwdelch` are macros.

```
deleteln( )
wdeleteln(win)
WINDOW *win;
```

The line under the cursor in the window is deleted. All lines below the current line are moved up one line. The bottom line of the window is cleared. The cursor position does not change. (This does not imply use of the hardware delete line feature.)

NOTE: `deleteln` is a macro.

```
getyx(win, y, x)
WINDOW *win;
int y, x;
```

The cursor position of the window is placed in the two integer variables `y` and `x`. This is implemented as a macro, so no & is necessary before the variables.

NOTE: `getyx` is a macro.

```
insch(ch)
chtype ch;
winsch(win, ch)
WINDOW *win;
chtype ch;
mvinsch(y, x, ch)
int y, x;
chtype ch;
mvwinsch(win, y, x, ch)
WINDOW *win;
int y, x;
chtype ch;
```

The character `ch` is inserted before the character under the cursor. All characters to the right are moved one space to the right, possibly losing the rightmost character on the line. The cursor position does not change, (after moving to `y, x`, if specified). (This does not imply use of the hardware insert character feature.)

NOTE: `insch, mvinsch,` and `mvwinsch` are macros.

```
insertln( )
winsertln(win)
WINDOW *win;
```

A blank line is inserted above the current line and the bottom line is lost. (This does not imply use of the hardware insert line feature.)

NOTE: `insertln` is a macro.

```
move(y, x)
wmove(win, y, x)
WINDOW *win;
int y, x;
```

The cursor associated with the window is moved to the given location. This does not move the physical cursor of the terminal until `refresh()` is called. The position specified is relative to the upper left corner of the window, which is (0,0).

NOTE: `move` is a macro.

```
overlay(srcwin, dstwin)
WINDOW *srcwin, *dstwin;
overwrite(srcwin, dstwin)
WINDOW *srcwin, *dstwin;
```

These routines overlay `srcwin` on top of `dstwin`, that is, all text in `srcwin` is copied into `dstwin`. `Scrwin` and `dstwin` are not required to be the same size. The copy starts at (0,0) on each window. The difference is that `overlay()` is non-destructive (blanks are not copied) while `overwrite()` is destructive.

```
printw(fmt [, arg] ...)
char *fmt;
wprintw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;
mvprintw(y, x, fmt [, arg] ...)
int y, x;
char *fmt;
mvwprintw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;
```

These routines are analogous to `printf`. The string which would be

output by `printf()` is instead output using `waddstr()` on the given window.

```
scroll(win)
WINDOW *win;
```

The window is scrolled up one line. This involves moving the lines in the window data structure. As an optimization, if the window is `stdscr` and the scrolling region is the entire window, the physical screen will be scrolled at the same time.

```
touchwin(win)
WINDOW *win;
```

Throw away all optimization information about which parts of the window have been touched, by pretending that the entire window has been drawn on. This is sometimes necessary when using overlapping windows, since a change to one window will affect the other window, but the records of which lines have been changed in the other window will not reflect the change.

Input

The following routines are used to obtain input from windows.

```
getch( )
wgetch(win)
WINDOW *win;
mvgetch(y, x)
int y, x;
mvwgetch(win, y, x)
WINDOW *win;
int y, x;
```

A character is read from the terminal associated with the window. In nodelay mode, if there is no input waiting, the value ERR is returned. In delay mode, the program will hang until the system passes text through to the program. Depending on the setting of `cbreak()`, this will be after one character, or after the first newline. Unless `noecho()` has been set, the character will also be echoed into the designated window.

If `keypad()` is `TRUE`, and a function key is pressed, the token for that function key will be returned instead of the raw characters. Possible function keys are defined in `<curses.h>` with integers beginning with 0401, whose names begin with `KEY_`. If a character is received that could be the beginning of a function key (such as escape), *curses* will set a timer. If the remainder of the sequence does not come in within the designated time, the character will be passed through, otherwise the function key value will be returned. For this reason, on many terminals, there will be a delay after a user presses the escape key before the escape is returned to the program. (Use by a programmer of the escape key for a single character function is discouraged.)

NOTE: getch, mvgetch, and mvwgetch are macros.

```
getstr(str)
char *str;
wgetstr(win, str)
WINDOW *win;
char *str;
mvgetstr(y, x, str)
int y, x;
char *str;
mvwgetstr(win, y, x, str)
WINDOW *win;
int y, x;
char *str;
```

A series of calls to getch() is made, until a newline and carriage return is received. The resulting value is placed in the area pointed at by the character pointer str. The user's erase and kill characters are interpreted.

NOTE: getstr, mvgetstr, and mvwgetstr are macros.

```
flushinp( )
```

Throws away any typeahead that has been typed by the user and has not yet been read by the program.

```
inch( )
winch(win)
WINDOW *win;
mvinch(y, x)
int y, x;
mvwinch(win, y, x)
WINDOW *win;
int y, x;
```

The character, of type chtype, at the current position in the named window is returned. If any attributes are set for that position, their values will be OR'ed into the value returned. The predefined constants A_CHARTEXT and A_ATTRIBUTES, defined in <curses.h>, can be used with the & (logical and) operator to extract the character or attributes alone.

NOTE: inch, winch, mvinch, and mvwinch are macros.

```
scanw(fmt [, arg] ...)
char *fmt;
wscanw(win, fmt [, arg] ...)
WINDOW *win;
char *fmt;
mvscanw(y, x, fmt [, arg] ...)
int y, x;
```

```
char *fmt;
mvwscanw(win, y, x, fmt [, arg] ...)
WINDOW *win;
int y, x;
char *fmt;
```

These routines correspond to `scanf`. `wgetstr()` is called on the window, and the resulting line is used as input for the scan.

Output Options Setting

These routines set options within *curses* that deal with output. All options are initially `FALSE`, unless otherwise stated. It is not necessary to turn these options off before calling `endwin()`.

```
clearok(win, bf)
WINDOW *win;
bool bf;
```

If `bf` is `TRUE`, the next call to `wrefresh()` with this window will clear the screen completely and redraw the entire screen from scratch. This is useful when the contents of the screen are uncertain, or in some cases for a more pleasing visual effect.

```
idlok(win, bf)
WINDOW *win;
bool bf;
```

If enabled (`bf` is `TRUE`), *curses* will consider using the hardware insert/delete line feature of terminals so equipped. If disabled (`bf` is `FALSE`), *curses* will very seldom use this feature. (The insert/delete character feature is always considered.) This option should be enabled only if the application needs insert/delete line, for example, for a screen editor. It is disabled by default because insert/delete line tends to be visually annoying when used in applications where it isn't really needed. If insert/delete line cannot be used, *curses* will redraw the changed portions of all lines.

```
leaveok(win, bf)
WINDOW *win;
bool bf;
```

Normally, the hardware cursor is left at the location of the window cursor being refreshed. This option allows the cursor to be left wherever the update happens to leave it. It is useful for applications where the cursor is not used, since it reduces the need for cursor motions. If possible, the cursor is made invisible when this option is enabled.

```
setscrreg(top, bot)
int top, bot;
wsetscrreg(win, top, bot)
WINDOW *win;
int top, bot;
```

These routines allow the user to set a software scrolling region in a window. `top` and `bot` are the line numbers of the top and bottom margin of the scrolling region. (Line 0 is the top line of the window.) If this option and `scrollok()` are enabled, an attempt to move off the bottom margin line will cause all lines in the scrolling region to scroll up one line. Only the text of the window is scrolled. (Note that this has nothing to do with use of a physical scrolling region capability in the terminal, like that in the VT100. If `idlok()` is enabled and the terminal has either a scrolling region or insert/delete line capability, they will probably be used by the output routines.)

```
scrollok(win, bf)
WINDOW *win;
bool bf;
```

This option controls what happens when the cursor of a window is moved off the edge of the window or scrolling region, either from a newline on the bottom line, or typing the last character of the last line. If disabled, the cursor is left on the bottom line. If enabled, `wrefresh()` is called on the window, and then the physical terminal and window are scrolled up one line. (Note that in order to get the physical scrolling effect on the terminal, is is also necessary to call `idlok()`.)

```
nl( )
nonl( )
```

These routines control whether newline is translated into carriage return and linefeed on output, and whether return is translated into newline on input. Initially, the translations do occur. By disabling these translations, *curses* is able to make better use of the linefeed capability, resulting in faster cursor motion.

NOTE: `nl` is a macro.

Input Options Setting

```
cbreak( )
nocbreak( )
```

These two routines put the terminal into and out of CBREAK mode. In this mode, characters typed by the user are immediately available to the program and erase/kill character processing is not performed. When out of this mode, the teletype driver will buffer characters typed until a newline or carriage return is typed. Interrupt and flow control characters are unaffected by this mode. Initially the terminal may or may not be in CBREAK mode, as it is inherited. Most interactive programs using *curses* will set this mode.

```
def_prog_mode( )
def_shell_mode( )
saveterm( )
```

Save the current terminal modes as the "program" (in *curses*) or "shell" (not in *curses*) state for use by the `reset_prog_mode()` and `reset_shell_mode()` routines. This is done automatically by `initscr()`.

NOTE: The `saveterm()` routine is being replaced by `def_prog_mode()`, which provides the same functionality. `Saveterm()` is included here for compatibility and is supported at level 2.

`echo( )`

`noecho( )`

These routines control whether characters typed by the user are echoed by `getch()` as they are typed. Initially, characters typed are echoed. Authors of most interactive programs prefer to do their own echoing in a controlled area of the screen, or not to echo at all, so they disable echoing. Echoing by the tty driver is always disabled.

```
intrflush(win, bf)
WINDOW *win;
bool bf;
```

If this option is enabled, when an interrupt key is pressed on the keyboard (interrupt, break, quit) all output in the tty driver queue will be flushed, giving the effect of faster response to the interrupt, but causing *curses* to have the wrong idea of what is on the screen. Disabling the option prevents the flush. The default for the option is inherited from the tty driver settings. The window argument is ignored.

```
keypad(win, bf)
WINDOW *win;
bool bf;
```

This option enables the keypad of the user's terminal. If enabled, the user can press a function key (such as an arrow key) and `getch()` will return a single value representing the function key, as in `KEY_LEFT`. (See **Function Keys** below.) If disabled, *curses* will not treat function keys specially and the program would have to interpret the escape sequences itself. If the keypad in the terminal can be turned on (made to transmit) and off (made to work locally), turning on this option will cause the terminal keypad to be turned on when `wgetch()` is called.

```
nodelay(win, bf)
WINDOW *win;
bool bf;
```

This option causes `getch()` to be a non-blocking call. If no input is ready, `getch()` will return `ERR`. If disabled, `getch()` will hang until a key is pressed.

`raw( )`

`noraw( )`

The terminal is placed into or out of raw mode. Raw mode is similar to CBREAK mode, in that characters typed are immediately passed through to the user program. The differences are that in RAW mode, the interrupt, quit, suspend and flow control characters are passed through uninterpreted, instead of generating a signal. The behavior of the BREAK key depends on other bits in the tty driver that are not set by *curses*.

`reset_prog_mode( )`

`reset_shell_mode( )`

`fixterm( )`

`resetterm( )`

Restore the terminal to "program" (in *curses*) or "shell" (out of *curses*) state. These are done automatically by `endwin()` and `doupdate()` after an `endwin()`, so they would normally not be called before.

NOTE: The `fixterm()` routine is being replaced by `reset_prog_mode()` and the `resetterm()` routine is being replaced by `reset_shell_mode()`. `fixterm()` and `resetterm()` are included here for compatibility and are supported at level 2.

`resetty( )`

`savetty( )`

These routines save and restore the state of the terminal modes. `savetty()` saves the current state in a buffer and `resetty()` restores the state to what it was at the last call to `savetty()`.

`typeahead(fd)`
`int fd;`

Curses does "line-breakout optimization" by looking for typeahead periodically while updating the screen. If input is found, the current update will be postponed until `refresh()` or `doupdate( )` is called again. This allows faster response to commands typed in advance. Normally, the input FILE pointer passed to `newterm()`, or `stdin` in the case that `initscr()` was used, will be used to do this typeahead checking. The `typeahead()` routine specifies that the file descriptor `fd` is to be used to check for typeahead instead. If `fd` is −1, then no typeahead checking will be done.

Environment Queries

`baudrate( )`

Returns the output speed of the terminal. The number returned is in bits per second, for example 9600, and is an integer.

`erasechar( )`

The user's current erase character is returned.

`has_ic( )`
True if the terminal has insert- and delete-character capabilities.

`has_il( )`
True if the terminal has insert- and delete-line capabilities, or can simulate them using scrolling regions. This might be used to check to see if it would be appropriate to turn on physical scrolling using `scrollok()`.

`killchar( )`
The user's current line kill character is returned.

`char *longname( )`
This routine returns a pointer to a static area containing a verbose description of the current terminal. The maximum length of a verbose description is 128 characters. It is defined only after the call to `initscr()` or `newterm()`. The area is overwritten by each call to `newterm()` and is not restored by `set_term()`, so the value should be saved between calls to `newterm()` if `longname()` is going to be used with multiple terminals.

Terminfo Level Routines

These low level routines must be called by programs that need to deal directly with the *terminfo* database to handle certain terminal capabilities, such as programming function keys. For all other functionality, *curses* routines are more suitable and their use is recommended.

Initially, `setupterm()` should be called. (Note that `setupterm()` is automatically called by `initscr()` and `newterm()`.) This will define the set of terminal dependent variables defined in TERMINFO(TI_ENV). The *terminfo* variables `lines` and `columns` are initialized by `setupterm()` as follows. If the environment variables LINES and COLUMNS exist, their values are used. If the above environment variables do not exist and the program is running in a window, the current window size is used. Otherwise, the values for `lines` and `columns` specified in the *terminfo* database are used.

The header files `<curses.h>` and `<term.h>` should be included to get the definitions for these strings, numbers, and flags. Parameterized strings should be passed through `tparm()` to instantiate them. All *terminfo* strings (including the output of `tparm()`) should be printed with `tputs()` or `putp()`. Before exiting, `reset_shell_mode()` should be called to restore the tty modes. Programs which use cursor addressing should output `enter_ca_mode` upon startup and should output `exit_ca_mode` before exiting (Programs desiring shell escapes should call `reset_shell_mode()` and output `exit_ca_mode` before the shell is called and should output `enter_ca_mode` and call `reset_prog_mode()` after returning from the shell.)

```
setupterm(term, fildes, errret)
char *term;
int fildes;
int *errret;
```

```
setterm(term)
char *term;
```

Read in the *terminfo* database, initializing the *terminfo* structures, but do not set up the output virtualization structures used by *curses*. The terminal type is the character string `term`; if `term` is null, the environment variable `TERM` will be used. All output is to file descriptor `fildes`. If `errret` is given, then `setupterm()` will return `OK` or `ERR` and store a status value in the integer pointed to by `errret`. A status of 1 in `errret` is normal, 0 means that the terminal could not be found, and −1 means that the *terminfo* database could not be found. If `errret` is NULL, `setupterm()` will print an error message upon finding an error and exit. Thus, the simplest call is `setupterm ((char *) 0, 1, (int *) 0)`, which uses all the defaults.

NOTE: The `setterm()` routine is being replaced by `setupterm()`. The call `setupterm(term, 1, (int *) 0)` provides the same functionality as `setterm(term)`. `setterm()` is included here for compatibility and is supported at level 2.

```
char *tparm(str, p1, p2, ..., p9)
char *str;
int p1, p2, p3, p4, p5, p6, p7, p8, p9;
```

Instantiate the string `str` with parms `pi`. A pointer is returned to the result of `str` with the parameters applied.

```
tputs(str, affcnt, putc)
char *str;
int affcnt;
int (*putc)();
```

Apply padding info to the string `str` and output it. `str` must be a *terminfo* string variable or the return value from `tparm()`, `tgetstr()`, or `tgoto()`. `affcnt` is the number of lines affected, or 1 if not applicable. `putc()` is a `putchar` like routine to which the characters are passed, one at a time.

```
putp(str)
char *str;
```

A routine that calls `tputs(str, 1, putchar)`.

```
vidputs(attrs, putc)
int attrs;
int (*putc)();
```

Output the string to put the terminal in the video attribute mode `attrs`, which is any combination of the attributes listed below. The characters are passed to the `putchar` like routine `putc`.

```
vidattr(attrs)
int attrs;
```
Like `vidputs()`, except that it outputs through `putchar`.

```
mvcur(oldrow, oldcol, newrow, newcol)
int oldrow, oldcol, newrow, newcol;
```
Low level cursor motion.

Termcap Compatibility Routines

These routines were included as a conversion aid for programs that use the termcap library. Their parameters are the same and the routines are emulated using the *terminfo* database.

```
tgetent(bp, name)
char *bp, *name;
```
Look up termcap entry for `name`. The emulation ignores the buffer pointer `bp`.

```
tgetflag(id)
char id[2];
```
Get the boolean entry for `id`.

```
tgetnum(id)
char id[2];
```
Get numeric entry for `id`.

```
char *tgetstr(id, area)
char id[2];
char **area;
```
Return the string entry for `id`. `tputs()` should be used to output the returned string.

```
char *tgoto(cap, col, row)
char *cap;
int col, row;
```
Instantiate the parameters into the given capability. The output from this routine is to be passed to `tputs()`.

```
tputs(str,affcnt,putc)
char *str;
int affcnt;
int (*putc)();
```
(See `tputs()` under **Terminfo Level Routines** above.)

Miscellaneous

```
unctrl(c)
chtype c;
```
This macro expands to a character string which is a printable representation of the character `c`. Control characters are displayed in the ^X

notation. Printing characters are displayed as is.

NOTE: `unctrl` is a macro, which is defined in `<unctrl.h>`.

`gettmode( )`
No-op.

NOTE: `gettmode()` is included here for compatibility and is supported at level 2.

Attributes

The following video attributes, defined in `<curses.h>`, can be passed to the routines `attron()`, `attroff()`, and `attrset()`.

A_STANDOUT	Terminal's best highlighting mode
A_UNDERLINE	Underlining
A_REVERSE	Reverse video
A_BLINK	Blinking
A_DIM	Half bright
A_BOLD	Extra bright or bold
A_CHARTEXT	Bit-mask to extract a character
A_ATTRIBUTES	Bit-mask to extract attributes

Function Keys

The following function keys, defined in `<curses.h>`, might be returned by `getch()` if `keypad()` has been enabled. Note that not all of these may be supported on a particular terminal if the terminal does not transmit a unique code when the key is pressed or the definition for the key is not present in the *terminfo* database.

Name	*Value*	*Key name*
KEY_BREAK	0401	Break key
KEY_DOWN	0402	The four arrow keys ...
KEY_UP	0403	
KEY_LEFT	0404	
KEY_RIGHT	0405	...
KEY_HOME	0406	Home key (upward+left arrow)
KEY_BACKSPACE	0407	Backspace
KEY_F0	0410	Function keys; space for 64 keys is reserved.
KEY_F(n)	(KEY_F0+(n))	
KEY_DL	0510	Delete line
KEY_IL	0511	Insert line
KEY_DC	0512	Delete character
KEY_IC	0513	Insert char or enter insert mode
KEY_EIC	0514	Exit insert char mode
KEY_CLEAR	0515	Clear screen

KEY_EOS	0516	Clear to end of screen
KEY_EOL	0517	Clear to end of line
KEY_SF	0520	Scroll 1 line forward
KEY_SR	0521	Scroll 1 line backwards (reverse)
KEY_NPAGE	0522	Next page
KEY_PPAGE	0523	Previous page
KEY_STAB	0524	Set tab
KEY_CTAB	0525	Clear tab
KEY_CATAB	0526	Clear all tabs
KEY_ENTER	0527	Enter or send
KEY_SRESET	0530	Soft (partial) reset
KEY_RESET	0531	Reset or hard reset
KEY_PRINT	0532	Print or copy
KEY_LL	0533	Home down or bottom (lower left) Keypad is arranged like this: A1 up A3 left B2 right C1 down C3
KEY_A1	0534	Upper left of keypad
KEY_A3	0535	Upper right of keypad
KEY_B2	0536	Center of keypad
KEY_C1	0537	Lower left of keypad
KEY_C3	0540	Lower right of keypad

RETURN VALUE

All routines return the integer `OK` upon successful completion and the integer `ERR` upon failure, unless otherwise noted in the preceding routine descriptions.

Routines that return pointers always return `(type *) NULL` on error.

SEE ALSO

TERMINFO(TI_ENV).

USAGE

Application Program.

The header file `<curses.h>` automatically includes the header files `<stdio.h>` and `<unctrl.h>`.

FUTURE DIRECTIONS

The *curses* routines will be enhanced to define additional function keys; and to support line drawing alternate character sets.

The `tgetstr(id, area)` routine will be enhanced to store the string entry returned for `id` in the buffer pointed to by `area` and advance `area`.

LEVEL

Level 1: All routines except `fixterm()`, `gettmode()`, `resetterm()`, `saveterm()`, `setterm()`, and the `termcap` compatibility routines.

Level 2: December 1, 1985 for `fixterm()`, `gettmode()`, `resetterm()`, `saveterm()`, `setterm()`, and the *termcap* compatibility routines.

Chapter 15
Commands and Utilities

NAME

tic — *terminfo* compiler

SYNOPSIS

```
tic [ -v[n] ] file
```

DESCRIPTION

The command `tic` translates a *terminfo* file from the source format into the compiled format. The results are placed in the directory `/usr/lib/terminfo`. The compiled format is necessary for use with the library routines described in CURSES(TI_LIB). The argument *file* contains one or more *terminfo* terminal descriptions in source format [see TERMINFO(TI_ENV)].

The option `-v` (verbose) causes `tic` to output trace information showing its progress. The optional integer *n* is a number from 1 to 10, inclusive, indicating the desired level of detail of information. If *n* is omitted, the default level is 1. If *n* is specified and greater than 1, the level of detail is increased.

The command `tic` compiles all *terminfo* descriptions in the given file. Each description in the file describes the capabilities of a particular terminal. When a **use=***entry_name* field is discovered in the terminal entry currently being compiled, `tic` duplicates the capabilities in *entry_name* for the current entry, with the exception of those capabilities that are explicitly defined in the current entry.

If the environment variable `TERMINFO` is set, the compiled results are placed there instead of `/usr/lib/terminfo`.

Total compiled entries cannot exceed 4096 bytes. The name field cannot exceed 128 bytes.

FILES

`/usr/lib/terminfo/?/*` Compiled terminal description database

SEE ALSO

CURSES(TI_LIB), TERMINFO(TI_ENV).

USAGE

Administrator.

When an entry, e.g. `entry_name_1`, contains a **use=***entry_name_2* field, any cancelled capabilities in *entry_name_2* must also appear in `entry_name_1` before **use=** for these capabilities to be cancelled in `entry_name_1`.

LEVEL

Level 1.

NAME

tput — query *terminfo* database

SYNOPSIS

`tput [ -Ttype ] capname`

DESCRIPTION

The command `tput` uses the *terminfo* database to make the values of terminal-dependent capabilities and information available to the shell [see SH(BU_CMD)]. The command `tput` outputs a string if the attribute is of type string, or an integer if the attribute is of type integer. If the attribute is of type boolean, `tput` simply sets the exit code (0 for TRUE if the terminal has the capability, 1 for FALSE if it does not), and produces no output.

`-Ttype` indicates the type of terminal. Normally this option is unnecessary, as the default is taken from the environment variable `TERM`.

`capname` indicates the attribute from the *terminfo* database. [See TERMINFO(TI_ENV).

EXAMPLES

`tput clear`
Echo clear-screen sequence for the current terminal.

`tput cols`
Print the number of columns for the current terminal.

`tput -T450 cols`
Print the number of columns for the 450 terminal.

```
bold=`tput smso`
offbold=`tput rmso`
```

Set the shell variables "bold" to begin standout mode sequence and "offbold" to end standout mode sequence for the current terminal. This might be followed by a prompt, e.g.:

```
echo "${bold}Name: ${offbold}\c"
```

`tput hc`
Set exit code to indicate if the current terminal is a hardcopy terminal.

FILES

`/usr/lib/terminfo/?/*`
Compiled terminal description database

RETURN VALUE

If `capname` is of type boolean, a value of 0 is returned for TRUE and 1 for FALSE.

If `capname` is of type string or integer, a value of 0 is returned upon successful completion.

Any other value returned indicates an error, for example, the specification of a bad `capname` (i.e., *terminfo* does not support a capability named `capname`).

SEE ALSO

STTY(AU_CMD), TERMINFO(TI_ENV).

USAGE

Application Program.

FUTURE DIRECTIONS

The command `tput` will be enhanced to support `capnames` of type string which require the specification of one or more parameters (which are the arguments to `capname`).

The command `tput` will be enhanced to output a terminal's initialization strings or reset strings through the specification of a single argument.

LEVEL

Level 1.

Indexes

General Index

S

T

U

Command and Function Index

R

S

T