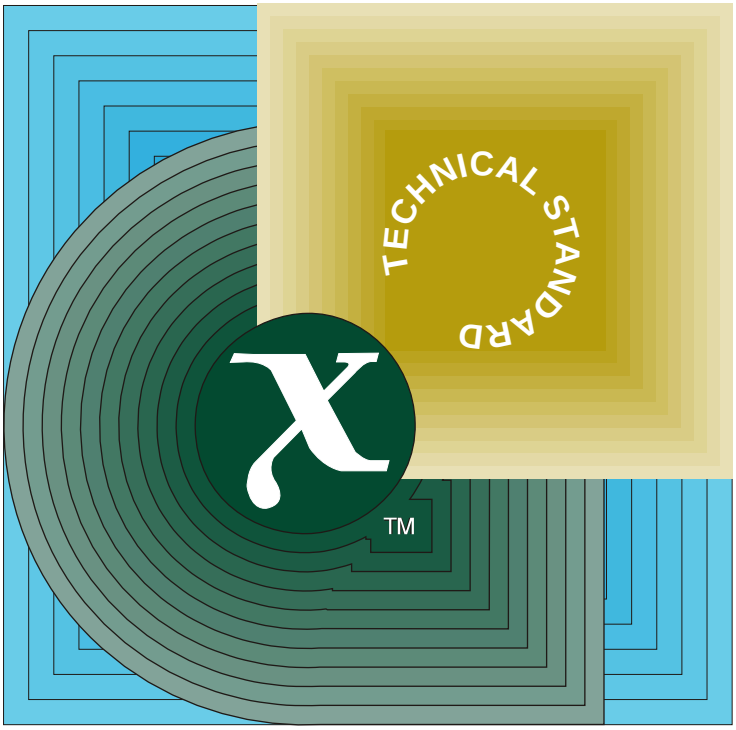
Technical Standard

Commands and Utilities
Issue 4, Version 2



THE Open GROUP

[This page intentionally left blank]

X/Open CAE Specification

Commands and Utilities

Issue 4, Version 2

X/Open Company Ltd.

X/Open CAE Specification

Commands and Utilities Issue 4, Version 2

ISBN: 1-85912-034-2

X/Open Document Number: C436

Published by X/Open Company Ltd., U.K.

Any comments relating to the material contained in this document may be submitted to X/Open at:

X/Open Company Limited
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

XoSpecs@xopen.co.uk

Contents

List of Tables

Preface

X/Open

X/Open is an independent, worldwide, open systems organisation supported by most of the world's largest information systems suppliers, user organisations and software companies. Its mission is to bring to users greater value from computing, through the practical implementation of open systems.

X/Open's strategy for achieving this goal is to combine existing and emerging standards into a comprehensive, integrated, high-value and usable open system environment, called the Common Applications Environment (CAE). This environment covers the standards, above the hardware level, that are needed to support open systems. It provides for portability and interoperability of applications, and so protects investment in existing software while enabling additions and enhancements. It also allows users to move between systems with a minimum of retraining.

X/Open defines this CAE in a set of specifications which include an evolving portfolio of application programming interfaces (APIs) which significantly enhance portability of application programs at the source code level, along with definitions of and references to protocols and protocol profiles which significantly enhance the interoperability of applications and systems.

The X/Open CAE is implemented in real products and recognised by a distinctive trade mark — the X/Open brand — that is licensed by X/Open and may be used on products which have demonstrated their conformance.

X/Open Technical Publications

X/Open publishes a wide range of technical literature, the main part of which is focussed on specification development, but which also includes Guides, Snapshots, Technical Studies, Branding/Testing documents, industry surveys, and business titles.

There are two types of X/Open specification:

- *CAE Specifications*

  CAE (Common Applications Environment) specifications are the stable specifications that form the basis for X/Open-branded products. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement an X/Open CAE specification can enjoy the benefits of a single, widely supported standard. In addition, they can demonstrate compliance with the majority of X/Open CAE specifications once these specifications are referenced in an X/Open component or profile definition and included in the X/Open branding programme.

  CAE specifications are published as soon as they are developed, not published to coincide with the launch of a particular X/Open brand. By making its specifications available in this way, X/Open makes it possible for conformant products to be developed as soon as is practicable, so enhancing the value of the X/Open brand as a procurement aid to users.

- *Preliminary Specifications*

  These specifications, which often address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations, are released in a controlled manner for the purpose of validation through implementation of products. A Preliminary specification is not a draft specification. In fact, it is as stable as X/Open can make it, and on publication has gone through the same rigorous X/Open development and review procedures as a CAE specification.

  Preliminary specifications are analogous to the *trial-use* standards issued by formal standards organisations, and product development teams are encouraged to develop products on the basis of them. However, because of the nature of the technology that a Preliminary specification is addressing, it may be untried in multiple independent implementations, and may therefore change before being published as a CAE specification. There is always the intent to progress to a corresponding CAE specification, but the ability to do so depends on consensus among X/Open members. In all cases, any resulting CAE specification is made as upwards-compatible as possible. However, complete upwards-compatibility from the Preliminary to the CAE specification cannot be guaranteed.

In addition, X/Open publishes:

- *Guides*

  These provide information that X/Open believes is useful in the evaluation, procurement, development or management of open systems, particularly those that are X/Open-compliant. X/Open Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming X/Open conformance.

- *Technical Studies*

  X/Open Technical Studies present results of analyses performed by X/Open on subjects of interest in areas relevant to X/Open's Technical Programme. They are intended to communicate the findings to the outside world and, where appropriate, stimulate discussion and actions by other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism for X/Open to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of an X/Open technical activity. Although at the time of its publication, there may be an intention to progress the activity towards publication of a Specification, Guide or Technical Study, X/Open is a consensus organisation, and makes no commitment regarding future development and further publication. Similarly, a Snapshot does not represent any commitment by X/Open members to develop any specific products.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision, in this case as the subject technology develops and to align with emerging associated international standards. X/Open makes a distinction between revised specifications which are fully backward compatible and those which are not:

- a new *Version* indicates that this publication includes all the same (unchanged) definitive information from the previous publication of that title, but also includes extensions or additional information. As such, it *replaces* the previous publication.

- a new *Issue* does include changes to the definitive information contained in the previous publication of that title (and may also include extensions or additional information). As such, X/Open maintains *both* the previous and new issue as current publications.

Corrigenda

Most X/Open publications deal with technology at the leading edge of open systems development. Feedback from implementation experience gained from using these publications occasionally uncovers errors or inconsistencies. Significant errors or recommended solutions to reported problems are communicated by means of Corrigenda.

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done either by email to the X/Open info-server or by checking the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information by email, send a message to info-server@xopen.co.uk with the following in the Subject line:

```
request corrigenda; topic index
```

This will return the index of publications for which Corrigenda exist.

This Document

This specification is one of a set of X/Open CAE Specifications (see above) defining the X/Open System Interface (XSI) Operating System requirements:

- System Interface Definitions, Issue 4, Version 2 (the **XBD** specification)
- Commands and Utilities, Issue 4, Version 2 (this document)
- System Interfaces and Headers, Issue 4, Version 2 (the **XSH** specification).

This document describes the commands and utilities offered to application programs on XSI-conformant systems. Readers are expected to be familiar with the **XBD** specification. This specification is structured as follows:

- Chapter 1 explains the status of the document and its relationship to formal standards. It also describes the defaults used by the utility descriptions in Chapter 3.
- Chapter 2 describes the command language used in XSI-conformant systems.
- Chapter 3 consists of manual pages for all utilities available on XSI-conformant systems.

Comprehensive references are available in the index.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names

  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (`...`) are used to show that additional arguments are optional.
- **`Bold fixed width`** font is used to identify brackets that surround optional items in syntax, **`[ ]`**, and to identify system output in interactive examples.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- Ranges of values are indicated with parentheses or brackets as follows:
  - — (a,b) means the range of all values from a to b, including neither a nor b
  - — [a,b] means the range of all values from a to b, including a and b
  - — [a,b) means the range of all values from a to b, including a, but not b
  - — (a,b] means the range of all values from a to b, including b, but not a
- Shading is used to identify extensions or warnings as detailed in Section 1.7.1 on page 8.

Trade Marks

AT&T® is a registered trade mark of AT&T in the U.S.A. and other countries.

HP® is a registered trade mark of Hewlett-Packard.

OSF™ is a trade mark of The Open Software Foundation, Inc.

UNIX® is a registered trade mark in the United States and other countries, licensed exclusively through X/Open Company Limited.

/usr/group® is a registered trade mark of UniForum, the International Network of UNIX System Users.

X/Open™ and the ‘‘X’’ device are trade marks of X/Open Company Limited.

Acknowledgements

X/Open gratefully acknowledges:

- AT&T for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The Institution of Electrical and Electronics Engineers, Inc. for permission to reproduce portions of its copyrighted IEEE Std 1003.2/D12, which have since become the corresponding portions of IEEE Std 1003.2-1992 and ISO/IEC 9945-2:1993, and also for permission to reproduce portions of IEEE Std P1003.1g/D4.
- The IEEE Computer Society's Portable Applications Standards Committee (PASC), whose Standards contributed to our work.
- The UniForum (formerly /usr/group) Technical Committee's Internationalization Subcommittee for work on internationalised regular expressions.
- The ANSI X3J11 Committees.
- Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc., for their work in developing the Single X/Open UNIX Extension and sponsoring it through the X/Open Direct Review (Fast-track) process.

Referenced Documents

The following documents are referenced in this specification:

ANS X3.9-1978
(Reaffirmed 1989) Programming Language FORTRAN.

ANSI C
ANS X3.159-1989, Programming Language C.

ANSI/IEEE Std 754-1985
Standard for Binary Floating-Point Arithmetic.

ANSI/IEEE Std 854-1987
Standard for Radix-Independent Floating-Point Arithmetic.

Ethernet
ISO 8802-3: 1990, Information Processing Systems — Local Area Networks — Part 3: Carrier Sense Multiple Access with Collision Detection (CSMA/CD) Access Method and Physical Layer Specifications.

ISO 4217
ISO 4217: 1987, Codes for the Representation of Currencies and Funds.

ISO 6937
ISO 6937: 1983, Information Processing — Coded Character Sets for Text Communication.

ISO 8601
ISO 8601: 1988, Data Elements and Interchange Formats — Information Interchange — Representation of Dates and Times.

ISO 8859-1
ISO 8859-1: 1987, Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Part 1: Latin Alphabet No. 1.

ISO/IEC 646
ISO/IEC 646: 1991, Information Processing — ISO 7-bit Coded Character Set for Information Interchange.

ISO/IEC 1539
ISO/IEC 1539: 1991, Information Technology — Programming Languages — Fortran.

ISO C
ISO/IEC 9899: 1990, Programming Languages — C (which is technically identical to ANS X3.159-1989, Programming Language C).

ISO POSIX-1
ISO/IEC 9945-1: 1990, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (which is identical to IEEE Std 1003.1-1990).

ISO POSIX-2
ISO/IEC 9945-2: 1993, Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities (which is identical to IEEE Std 1003.2-1992).

POSIX.1
IEEE Std 1003.1-1988, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language].

SVID Issue 1
System V Interface Definition (Spring 1985 - Issue 1).

SVID Issue 2
System V Interface Definition (Spring 1986 - Issue 2).

System V Release 2.0

— UNIX System V Release 2.0 Programmer's Reference Manual (April 1984 - Issue 2).

— UNIX System V Release 2.0 Programming Guide (April 1984 - Issue 2).

The following X/Open documents are referenced in this specification.

Internationalisation Guide, Version 2
X/Open Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-859120-02-4, G304).

Issue 1
X/Open Portability Guide, July 1985 (ISBN: 0-444-87839-4).

Issue 2
See **XCU, Issue 2**.

Issue 3
See **XCU, Issue 3**.

Issue 4
See **XCU, Issue 4**.

Issue 4, Version 2
See **XCU, Issue 4, Version 2**.

Migration Guide
X/Open Guide, July 1992, XPG3-XPG4 Base Migration Guide (ISBN: 1-872630-49-9, G204).

XBD, Issue 4
X/Open CAE Specification, July 1992, System Interface Definitions, Issue 4 (ISBN: 1-872630-46-4, C204).

XBD, Issue 4, Version 2
X/Open CAE Specification, August 1994, System Interface Definitions, Issue 4, Version 2 (ISBN: 1-85912-036-9, C434).

XCU, Issue 2
X/Open Portability Guide, Volume 1, January 1987, XVS Commands and Utilities (ISBN: 0-444-70174-5).

XCU, Issue 3
X/Open Specification, 1988, 1989, February 1992, Commands and Utilities, Issue 3 (ISBN: 1-872630-36-7, C211); this specification was formerly X/Open Portability Guide, Volume 1, January 1989 XSI Commands and Utilities (ISBN: 0-13-685835-X, XO/XPG/89/002).

XCU, Issue 4
X/Open CAE Specification, July 1992, Commands and Utilities, Issue 4 (ISBN: 1-872630-48-0, C203).

XCU, Issue 4, Version 2
X/Open CAE Specification, August 1994, Commands and Utilities, Issue 4, Version 2 (ISBN: 1-85912-034-2, C436). (This document.)

XNFS

X/Open CAE Specification, October 1992, Protocols for X/Open Interworking: XNFS, Issue 4 (ISBN: 1-872630-66-9, C218).

XPG4

X/Open Systems and Branded Products: XPG4, July 1992 (ISBN: 1-872630-52-9, X924).

XSH, Issue 4

X/Open CAE Specification, July 1992, System Interfaces and Headers, Issue 4 (ISBN: 1-872630-47-2, C202).

XSH, Issue 4, Version 2

X/Open CAE Specification, August 1994, System Interfaces and Headers, Issue 4, Version 2 (ISBN: 1-85912-037-7, C435).

Chapter 1

Introduction

1.1 Overview

This document defines the shell command language and the utilities provided by the X/Open System Interface (XSI). The utilities are accessed using commands given to command interpreters supporting the shell command language. The system interfaces and headers (described in X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**) and the utilities are jointly known as services to application programs. No particular restrictions are imposed on the way in which the services are implemented.

The utilities are defined in terms of their interface as seen from the *sh* command interpreter. Alternative interfaces are available to application programs through one of the *exec* functions, and the *popen*( ) and *system*( ) interfaces, all of which are described in X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2**.

1.2 Conformance

An implementation conforming to this document shall meet the following criteria:

- The system shall provide all the utilities described in this document with all the functionality defined, subject to the following:
  - — Optional utilities listed in Section 1.3.1 on page 2 need not be provided.
  - — Development utilities listed in Section 1.3.2 on page 2 need not be provided.
  - — FORTRAN utilities listed in Section 1.3.3 on page 3 need not be provided.
  - — Possibly unsupportable utilities listed in Section 1.3.4 on page 3 need not be provided.
  - — Withdrawn utilities listed in Section 1.4.4 on page 5 need not be provided.
  - — Within the utilities that are provided, functionality marked by the codes OF, OP, PI, or UN (see Section 1.7 on page 8) need not be provided.
- The system may provide one or more of the following:
  - — optional utilities (as a group) listed in Section 1.3.1 on page 2
  - — development utilities (as a group) listed in Section 1.3.2 on page 2
  - — the FORTRAN77 compiler listed in Section 1.3.3 on page 3.

  When an implementation claims that an optional or possibly unsupportable facility is provided, all of its constituent parts shall be provided and shall comply with the specification.

  When an implementation claims that a development system is provided, all of the utilities marked **DEVELOPMENT** and listed in Section 1.3.2 on page 2, except *dis*, shall be provided and shall comply with the specification.

  When an implementation claims that a FORTRAN system is provided, the utility marked **FORTRAN** and listed in Section 1.3.3 on page 3 shall be provided and shall comply with the specification.

Note: To determine whether an implementation supports optional, development, FORTRAN or possibly unsupportable utilities, refer to the implementation's Conformance Statement.

- The system may provide additional or enhanced utilities and facilities not required by this specification, provided that such additions or enhancements do not affect the behaviour of an application that requires only the facilities described in this document.

An implementation conforming to this document depends on the environment provided by the system interfaces and headers specified in the referenced **XSH** specification. For further information, refer to the **XSH** specification, **Section 1.2**, **Conformance**, the referenced **XPG4** document and the implementation's Conformance Statement.

1.2.1 Symbolic Links

UX The definition of symbolic links in the **XBD** specification, **Chapter 2**, **Glossary** is new in Issue 4, Version 2. An implementation need not support symbolic links to be conformant with this document. (See the **XSH** specification, **Section 1.2, Conformance** for the implications on conformance with the **XSH** specification.)

The definition of pathname resolution in the **XBD** specification, **Chapter 2**, **Glossary** indicates the effects of symbolic links. However, many utilities that manipulate files may manipulate symbolic links. Use of these utilities in this context (that is, when the last component of the pathname is a symbolic link) produces unspecified effects. In addition, if any utility encounters a symbolic link after pathname resolution has been completed, the effects are unspecified.

For future directions in this regard, see Section 1.6.1 on page 7.

1.3 Status of Interfaces

1.3.1 Optional

The *dis* utility is the only utility marked as **OPTIONAL** in this document.

1.3.2 Development

Utilities marked in this document as **DEVELOPMENT** are:

Development Utilities			
admin	*get*	*nm*	*strip*
cflow	*lex*	*prs*	*unget*
ctags	*lint*	*rmdel*	*val*
cxref	*m4*	*sact*	*what*
delta	*make*	*sccs*	*yacc*
dis			

1.3.3 FORTRAN

The *fort77* FORTRAN compiler is the only utility marked as **FORTRAN** in this document.

1.3.4 Possibly Unsupportable

Utilities that are possibly unsupportable in this document are marked as UN; they are:

Possibly Unsupportable Utilities			
cancel	*lpstat*	*uulog*	*uupick*
cu	*sdb*	*uuname*	*uuto*

1.4 Changes from Issue 3

1.4.1 Summary of Changes

The following list summarises the major changes that have been made in this document since Issue 3.

- Most utilities have been aligned with the ISO/IEC 9945-2:1993 standard (see Section 1.6 on page 7). The subdivision of the standard into optional components is generally followed, with the exceptions that the User Portability Extension is a mandatory feature of the XSI, and development options have been grouped together more closely.
- Utilities and features not defined by the ISO/IEC 9945-2:1993 standard are identified as extensions (see Section 1.7 on page 8). The more complete POSIX style of interface description is also used for these utilities.
- Some of the utilities have been updated to reflect general POSIX interface changes (such as the Utility Syntax Guidelines, definitions and limits).
- Some utilities are marked **TO BE WITHDRAWN** because a utility of similar functionality has been added.
- Some additional utilities are included; these are listed in the table in Section 1.4.2.

Although most existing shell and utility applications will continue to operate effectively on XSI-conformant systems, there are a number of changes that may be required to prepare fully portable, internationalised applications. These are described in the companion volume, the referenced **Migration Guide**. Because of the existence of the **Migration Guide** and the large number of interface and descriptive enhancements associated with the standards alignment, the **CHANGE HISTORY** sections in Chapter 3 do not offer detailed lists of technical changes.

1.4.2 New Features Added in Issue 4

The utilities first introduced in Issue 4 are as follows:

New Utilities			
alias	*asa*	*bg*	*c89*
cksum	*command*	*compress*	*expand*
fc	*fg*	*fold*	*fort77*
getconf	*getopts*	*head*	*jobs*
locale	*localedef*	*logger*	*more*
nice	*pathchk*	*pax*	*print*
renice	*sccs*	*strings*	*talk*
tput	*unalias*	*uncompress*	*unexpand*
uudecode	*uuencode*	*zcat*	

The *newgrp* and *mailx* utilities are changed from optional to mandatory in this issue.

1.4.3 To Be Withdrawn

Some of the utilities in this issue are marked TO BE WITHDRAWN. Various factors may have contributed to the decision to withdraw a utility. In all cases, the reasons for withdrawal of a utility are documented on the relevant pages.

If a migration path exists, advice is given to application developers regarding alternative means of obtaining similar functionality. This information may be found in the **APPLICATION USAGE** sections on the relevant pages.

Interfaces marked TO BE WITHDRAWN shall comply with this document's requirements. However, they will be marked WITHDRAWN in a future issue of this document. Interfaces marked WITHDRAWN may still exist on conformant implementations.

Application writers should not use functionality marked TO BE WITHDRAWN.

Interfaces marked in this issue to be withdrawn in the next issue are:

To Be Withdrawn			
calendar	*dircmp*	*mail*	*spell*
cc	*dis*	*pack*	*sum*
col	*line*	*pcat*	*tar*
cpio	*lint*	*pg*	*unpack*

1.4.4 Withdrawn

Withdrawn interfaces may still exist on conformant implementations. However, they will not appear in the next issue of this document. Application writers should not use functionality marked WITHDRAWN.

If a migration path exists, advice is given to application developers regarding alternative means of obtaining similar functionality. This information may be found in the **APPLICATION USAGE** sections on the relevant pages.

The following utilities are marked WITHDRAWN in this document:

Withdrawn	
banner	*sdb*
chroot	*wall*
red	

1.4.5 Issue 4, Version 2

The primary effect of Issue 4, Version 2 is to include in the **XSH** specification material on traditional interfaces on UNIX-based systems. This material is marked X/OPEN UNIX or flagged with the UX margin legend. In this **XCU** specification, the effect of this change is new information for the *c89* and *cc* utilities, also flagged with the UX margin legend.

1.5 Terminology

The following terms are used in this specification:

can
This describes a permissible optional feature or behaviour available to the user or application; all systems support such features or behaviour as mandatory requirements.

implementation-dependent
The value or behaviour is not consistent across all implementations. The provider of an implementation normally documents the requirements for correct program construction and correct data in the use of that value or behaviour. When the value or behaviour in the implementation is designed to be variable or customisable on each instantiation of the system, the provider of the implementation normally documents the nature and permissible ranges of this variation. Applications that are intended to be portable must not rely on implementation-dependent values or behaviour.

may
With respect to implementations, the feature or behaviour is optional. Applications should not rely on the existence of the feature. To avoid ambiguity, the reverse sense of *may* is expressed as *need not*, instead of *may not*.

must
This describes a requirement on the application or user.

obsolescent
Certain features are *obsolescent*, which means that they may be considered for withdrawal in future revisions of this document. They are retained in this version because of their widespread use. Their use in new applications is discouraged.

should
With respect to implementations, the feature is recommended, but it is not mandatory. Applications should not rely on the existence of the feature.

With respect to users or applications, the word means recommended programming practice that is necessary for maximum portability.

undefined
A value or behaviour is undefined if this document imposes no portability requirements on applications for erroneous program constructs or erroneous data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application using such behaviour is not fully portable to all systems.

unspecified
A value or behaviour is unspecified if this document imposes no portability requirements on applications for correct program construct or correct data. Implementations may specify the result of using that value or causing that behaviour, but such specifications are not guaranteed to be consistent across all implementations. An application requiring a specific behaviour, rather than tolerating any behaviour when using that functionality, is not fully portable to all systems.

will
This means that the behaviour described is a requirement on the implementation and applications can rely on its existence.

1.6 Relationship to Formal Standards

Great care has been taken to ensure that this document is fully aligned with the following formal standards:

- ISO / IEC 9945-1: 1990 (ISO POSIX-1)
- ISO / IEC 9945-2:1993 (ISO POSIX-2)
- ISO / IEC 9899: 1990 (ISO C).

Any conflict between this document and any of these standards is unintentional. This document defers to the formal standards, which X / Open recognises as superior. In particular, from time to time, when ambiguities are found in the formal standards, the responsible bodies will make interpretations of them, whose findings become binding on the standard. Where, as the result of such an interpretation, or for any other reason, any of these formal standards are found to conflict with this document, XSI-conformant systems are required to behave in the manner defined either by the formal standard or by this document. Application writers should clearly avoid depending exclusively on either behaviour in such cases; the list of all conflicts found since publication of this document is available on request. (See page ii for how to contact X / Open.)

1.6.1 Relationship to Emerging Formal Standards

A future edition of this document will be fully-aligned with the emerging IEEE 1003.2b standard. This will fully specify the behaviour of utilities in reference to symbolic links.

1.7 Portability

Some of the utilities in this document and functions in X/Open CAE Specification, **System Interfaces and Headers**, **Issue 4, Version 2** describe functionality that might not be fully portable to systems based on the ISO/IEC 9945-2:1993 standard or the ISO POSIX-1 standard. Where enhanced or reduced functionality is specified, the text is shaded and a code in the margin identifies the nature of the extension or warning (see Section 1.7.1). For maximum portability, an application should avoid such functionality.

Unless the primary task of a utility is to produce textual material on its standard output, application developers should not rely on the format or content of any such material that may be produced. Where the primary task *is* to provide such material, but the output format is incompletely specified, the description is marked. Application developers are warned not to expect that the output of such an interface on one system will be any guide to its behaviour on another system.

1.7.1 Codes

The codes and their meanings are as follows:

EX Extension.
The functionality described is an extension to the standards referenced above. Application writers may confidently make use of an extension as it will be supported on all XSI-conformant systems. These extensions are designed not to conflict with the published standards.

If an entire **SYNOPSIS** section is shaded and marked with one EX, all the functionality described in that entry is an extension.

Some behaviour which is allowed to be optional in the formal standards is mandated on XSI-conformant systems. Such behaviours (for example, those dependent on the availability of job control) may not be individually marked as extensions, but the mandatory nature of the feature is marked as an extension where the option is described, typically in the header file where the corresponding symbolic constant is defined.

FIPS FIPS Extension.
The **Federal Information Processing Standards (FIPS)** are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST). Where extensions have been made in order to align with the FIPS requirements, they have the special mark shown here, and appear in the index under FIPS alignment (as well as under EX).

The following extensions are required by FIPS 151-2:

- The implementation will support {_POSIX_CHOWN_RESTRICTED}.
- The limit {NGROUPS_MAX} will be greater than or equal to 8.
- The implementation will support the setting of the group ID of a file (when it is created) to that of the parent directory.
- The implementation will support {_POSIX_SAVED_IDS}.
- The implementation will support {_POSIX_VDISABLE}.
- The implementation will support {_POSIX_JOB_CONTROL}.
- The implementation will support {_POSIX_NO_TRUNC}.
- The *read*( ) call returns the number of bytes read when interrupted by a signal and will not return –1.

- The *write*( ) call returns the number of bytes written when interrupted by a signal and will not return –1.
- In the environment for the login shell, the environment variables *LOGNAME* and *HOME* will be defined and have the properties described in Chapter 5 of X/Open CAE Specification, **System Interface Definitions**, **Issue 4, Version 2**.
- The value of {CHILD_MAX} will be greater than or equal to 25.
- The value of {OPEN_MAX} will be greater than or equal to 20.
- The implementation will support the functionality associated with the symbols CS7, CS8, CSTOPB, PARODD and PARENB defined in **<termios.h>**.

JC Job Control Extension.
Job control is an optional feature in the operating system described by the ISO POSIX-1 standard, but it is supported by all XSI-conformant systems. When interfaces rely on this extension, they have the special mark shown here and appear in the index under JC (in addition to being under EX).

OB Obsolescent.
Some of the interfaces describe functionality that is obsolescent. Although these are fully portable to all current XSI-conformant systems they may be withdrawn in future issues.

OF Output format incompletely specified.
The format of the output produced by the utility is not fully specified. It is therefore not possible to post-process this output in a consistent fashion. Typical problems include unknown length of strings and unspecified field delimiters.

OP Dependent on optional service in XSI.
Typical implementations depend on an optional service and the functionality affected need not be present if the optional service is not supported.

PI The behaviour cannot be guaranteed to be consistent.
It is not possible to guarantee that the interface behaves in the same way on all XSI-conformant systems. This is the case if it provides functionality that is system-defined or system-specific. Options that are used to *select* alternative forms of system-specific behaviour are not marked, as it is clear from their descriptions that their use is inherently non-portable.

UN Possibly unsupportable feature.
It need not be possible to implement the required functionality (as defined) on all XSI-conformant systems and the functionality need not be present. This may, for example, be the case where the XSI-conformant system is hosted and the underlying system provides the service in an alternative way.

UX X/Open UNIX Extension
The material relates to interfaces included to provide portability for applications originally written to be compiled on UNIX and UNIX-based operating systems. Therefore, the features described may not be present on systems that conform to XPG4 or to earlier XPG releases. The relevant reference manual pages may provide additional or more specific portability warnings about use of the material.

If an entire **SYNOPSIS** section is shaded and marked with one UX, all the functionality described in that entry is an extension.

The material on pages labelled X/OPEN UNIX and the material flagged with the UX margin legend is available only in cases where the _XOPEN_UNIX version test macro is defined.

Withdrawal of Interfaces

Any interface (an entire utility, function or merely a feature) marked with one of the warning codes OB, PI or UN is subject to being withdrawn in a future issue. In these cases, the interface may be taken immediately to the WITHDRAWN state, without the usual TO BE WITHDRAWN step in an intermediate issue. For maximum portability, an application should avoid such functionality.

1.8 Grammar Conventions

Portions of this document are expressed in terms of a special grammar notation. It is used to portray the complex syntax of certain program input. The grammar is based on the syntax used by the *yacc* utility. However, it does not represent fully functional *yacc* input, suitable for program use; the lexical processing and all semantic requirements are described only in textual form. The grammar is not based on source used in any traditional implementation and has not been tested with the semantic code that would normally be required to accompany it. Furthermore, there is no implication that the partial *yacc* code presented represents the most efficient, or only, means of supporting the complex syntax within the utility. Implementations may use other programming languages or algorithms, as long as the syntax supported is the same as that represented by the grammar.

The following typographical conventions are used in the grammar; they have no significance except to aid in reading.

- The identifiers for the reserved words of the language are shown with a leading capital letter. (These are terminals in the grammar. Examples: **While**, **Case**.)
- The identifiers for terminals in the grammar are all named with upper-case letters and underscores. Examples: **NEWLINE**, **ASSIGN_OP**, **NAME**.
- The identifiers for non-terminals are all lower-case.

1.9 Utility Description Defaults

This section describes all of the subsections used within the utility descriptions, including:

- intended usage of the section
- global defaults that affect all the standard utilities
- the meanings of notations used in the standard that are specific to individual utility sections.

SYNOPSIS

The **SYNOPSIS** section summarises the syntax of the calling sequence for the utility, including options, option-arguments and operands. Standards for utility naming are described in **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**; for describing the utility's arguments in **XBD** specification, **Section 10.1**, **Utility Argument Syntax**.

DESCRIPTION

The **DESCRIPTION** section describes the actions of the utility. If the utility has a very complex set of subcommands or its own procedural language, an **EXTENDED DESCRIPTION** section is also provided. Most explanations of optional functionality are omitted here, as they are usually explained in the **OPTIONS** section.

Some utilities in this document are described in terms of functionality equivalent to the **XSH** specification. When specific functions are cited, the underlying operating system provides equivalent functionality and all side effects associated with successful execution of the function. The treatment of errors and intermediate results from the individual functions cited are generally not specified by this document. See the utility's **EXIT STATUS** and **CONSEQUENCES OF ERRORS** sections for all actions associated with errors encountered by the utility.

OPTIONS

The **OPTIONS** section describes the utility options and option-arguments, and how they modify the actions of the utility. Standard utilities that have options either fully comply with the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** or describe all deviations. Apparent disagreements between functionality descriptions in the **OPTIONS** and **DESCRIPTION** (or **EXTENDED DESCRIPTION**) sections are always resolved in favour of the **OPTIONS** section.

Each **OPTIONS** section that uses the phrase ''The ... utility supports the Utility Syntax Guidelines ...'' refers only to the use of the utility as specified by this document; implementation extensions should also conform to the guidelines, but may allow exceptions for historical practice.

Unless otherwise stated in the utility description, when given an option unrecognised by the implementation, or when a required option-argument is not provided, standard utilities will issue a diagnostic message to standard error and exit with a non-zero exit status.

EX All utilities in this document are capable of processing arguments using 8-bit transparency.

Default Behaviour: When this section is listed as ''None'', it means that the implementation need not support any options. Standard utilities that do not accept options, but that do accept operands, will recognise −− as a first argument to be discarded.

The requirement for recognising −− is because portable applications need a way to shield their operands from any arbitrary options that the implementation may provide as an extension. For example, if the standard utility *foo* is listed as taking no options,

and the application needed to give it a pathname with a leading hyphen, it could safely do it as:

```
foo -- -myfile
```

and avoid any problems with –**m** used as an extension.

OPERANDS

The **OPERANDS** section describes the utility operands, and how they affect the actions of the utility. Apparent disagreements between functionality descriptions in the **OPERANDS** and **DESCRIPTION** (or **EXTENDED DESCRIPTION**) sections are always resolved in favour of the **OPERANDS** section.

If an operand naming a file can be specified as ''-'', which means to use the standard input instead of a named file, this is explicitly stated in this section. Unless otherwise stated, the use of multiple instances of ''-'' to mean standard input in a single command produces unspecified results.

Unless otherwise stated, the standard utilities that accept operands will process those operands in the order specified in the command line.

Default Behaviour: When this section is listed as ''None'', it means that the implementation need not support any operands.

STDIN

The **STDIN** section describes the standard input of the utility. This section is frequently merely a reference to the following section, as many utilities treat standard input and input files in the same manner. Unless otherwise stated, all restrictions described in **INPUT FILES** apply to this section as well.

Use of a terminal for standard input can cause any of the standard utilities that read standard input to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

The specified standard input format of the standard utilities does not depend on the existence or value of the environment variables defined in this document, except as provided by this document.

Default Behaviour: When this section is listed as ''Not used,'' it means that the standard input will not be read when the utility is used as described by this document.

INPUT FILES

The **INPUT FILES** section describes the files, other than the standard input, used as input by the utility. It includes files named as operands and option-arguments as well as other files that are referred to, such as startup and initialisation files, databases, etc. Commonly-used files are generally described in one place and cross-referenced by other utilities.

EX All utilities in this document are capable of processing input files using 8-bit transparency.

When a standard utility reads a seekable input file and terminates without an error before it reaches end-of-file, the utility will ensure that the file offset in the open file description is properly positioned just past the last byte processed by the utility. For files that are not seekable, the state of the file offset in the open file description for that file is unspecified. A portable application cannot assume that the following three commands are equivalent:

```
tail -n +2 file
(sed -n 1q; cat) < file
cat file | (sed -n 1q; cat)
```

The second command is equivalent to the first only when the file is seekable. The third command leaves the file offset in the open file description in an unspecified state. Other utilities, such as *head*, *read* and *sh*, have similar properties.

Some of the standard utilities, such as filters, process input files a line or a block at a time and have no restrictions on the maximum input file size. Some utilities may have size limitations that are not as obvious as file space or memory limitations. Such limitations should reflect resource limitations of some sort, not arbitrary limits set by implementors. Implementations will document those utilities that are limited by constraints other than file system space, available memory and other limits specifically cited by this document, and identify what the constraint is and indicate a way of estimating when the constraint would be reached. Similarly, some utilities descend the directory tree (recursively). Implementations will also document any limits that they may have in descending the directory tree that are beyond limits cited by this document.

When an input file is described as a *text file*, the utility produces undefined results if given input that is not from a text file, unless otherwise stated. Some utilities (for example, *make*, *read*, *sh*) allow for continued input lines using an escaped <newline> convention; unless otherwise stated, the utility need not be able to accumulate more than {LINE_MAX} bytes from a set of multiple, continued input lines. Thus, for a portable application the total of all the continued lines in a set cannot exceed {LINE_MAX}. If a utility using the escaped <newline> convention detects an end-of-file condition immediately after an escaped <newline>, the results are unspecified.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See **XBD** specification, **Chapter 3**, **File Format Notation** for a description of this notation. The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since <blank> characters can legitimately be included in some of the fields described by the standard utilities, particularly in locales other than the POSIX Locale, this intent is not always realised.

Default Behaviour: When this section is listed as ''None'', it means that no input files are required to be supplied when the utility is used as described by this document.

ENVIRONMENT VARIABLES

The **ENVIRONMENT VARIABLES** section lists what variables affect the utility's execution.

The entire manner in which environment variables described in this document affect the behaviour of each utility is described in the **ENVIRONMENT VARIABLES** section
EX for that utility, in conjunction with the global effects of the *LANG*, *LC_ALL* and *NLSPATH* environment variables described in **XBD** specification, **Chapter 6**, **Environment Variables**. The existence or value of environment variables described in

this document will not otherwise affect the specified behaviour of the standard utilities. Any effects of the existence or value of environment variables not described by this document upon the standard utilities are unspecified.

For those standard utilities that use environment variables as a means for selecting a utility to execute (such as *CC* in *make*), the string provided to the utility is subjected to the path search described for *PATH* in the **XBD** specification, **Chapter 6**, **Environment Variables**.

EX All utilities in this document are capable of processing environment variable names and values using 8-bit transparency.

Default Behaviour: When this section is listed as ''None'', it means that the behaviour of the utility is not directly affected by environment variables described by this document when the utility is used as described by this document.

ASYNCHRONOUS EVENTS

The **ASYNCHRONOUS EVENTS** section lists how the utility reacts to such events as signals and what signals are caught.

Default Behaviour: When this section is listed as ''Default'', or it refers to ''the standard action for all other signals; see Section 1.9 on page 11'' it means that the action taken as a result of the signal is one of the following:

1. The action is that inherited from the parent according to the rules of inheritance of signal actions defined in the **XSH** specification.
2. When no action has been taken to change the default, the default action is that specified by the **XSH** specification.
3. The result of the utility's execution is as if default actions had been taken.

A utility is permitted to catch a signal, perform some additional processing (such as deleting temporary files), restore the default signal action (or action inherited from the parent process) and resignal itself.

STDOUT

The **STDOUT** section describes the standard output of the utility. This section is frequently merely a reference to the following section, **OUTPUT FILES**, because many utilities treat standard output and output files in the same manner.

Use of a terminal for standard output may cause any of the standard utilities that write standard output to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See the **XBD** specification, **Chapter 3**, **File Format Notation** for a description of this notation.

The specified standard output of the standard utilities does not depend on the existence or value of the environment variables defined in this document, except as provided by this document.

Some of the standard utilities describe their output using the verb *display*, defined in the **XBD** specification, **Chapter 2**, **Glossary**. Output described in the **STDOUT** sections of such utilities may be produced using means other than standard output. When standard output is directed to a terminal, the output described will be written directly to the terminal. Otherwise, the results are undefined.

Default Behaviour: When this section is listed as ‘‘Not used’’, it means that the standard output will not be written when the utility is used as described by this document.

STDERR

The **STDERR** section describes the standard error output of the utility. Only those messages that are purposely sent by the utility are described.

Use of a terminal for standard error may cause any of the standard utilities that write standard error output to stop when used in the background. For this reason, applications should not use interactive features in scripts to be placed in the background.

The format of diagnostic messages for most utilities is unspecified, but the language and cultural conventions of diagnostic and informative messages whose format is
EX unspecified by this standard should be affected by the setting of *LC_MESSAGES* and *NLSPATH*.

The specified standard error output of standard utilities does not depend on the existence or value of the environment variables defined in this document, except as provided by this document.

Default Behaviour: When this section is listed as ‘‘Used only for diagnostic messages,’’ it means that, unless otherwise stated, the diagnostic messages are sent to the standard error only when the exit status is non-zero and the utility is used as described by this document.

When this section is listed as ‘‘Not used’’, it means that the standard error will not be used when the utility is used as described in this document.

This section does not describe error messages that refer to incorrect operation of the utility. Consider a utility that processes program source code as its input. This section is used to describe messages produced by a correctly operating utility that encounters an error in the program source code on which it is processing. However, a message indicating that the utility had insufficient memory in which to operate would not be described.

Some compilers have traditionally produced warning messages without returning a non-zero exit status; these are specifically noted in their sections. Other utilities are expected to remain absolutely quiet on the standard error if they want to return zero, unless the implementation provides some sort of extension to increase the verbosity or debugging level.

OUTPUT FILES

The **OUTPUT FILES** section describes the files created or modified by the utility. Temporary or system files that are created for internal usage by this utility or other parts of the implementation (for example, spool, log and audit files) are not described in this, or any, section. The utilities creating such files and the names of such files are unspecified. If applications are written to use temporary or intermediate files, they should use the *TMPDIR* environment variable, if it is set and represents an accessible directory, to select the location of temporary files.

Temporary files used by the standard utilities are named so that different utilities or multiple instances of the same utility can operate simultaneously without regard to their working directories, or any other process characteristic other than process ID. There are two exceptions to this rule:

1. Resources for temporary files other than the name space (for example, disk space, available directory entries, or number of processes allowed) are not guaranteed.
2. Certain standard utilities generate output files that are intended as input for other utilities, (for example, *lex* generates **lex.yy.c**) and these cannot have unique names. These cases are explicitly identified in the descriptions of the respective utilities.

Any temporary file created by the implementation is removed by the implementation upon a utility's successful exit, exit because of errors, or before termination by any of the SIGHUP, SIGINT or SIGTERM signals, unless specified otherwise by the utility description.

Receipt of the SIGQUIT signal should generally cause termination (unless in some debugging mode) that would bypass any attempted recovery actions.

Record formats are described in a notation similar to that used by the C-language function, *printf*( ). See the **XBD** specification, **Chapter 3**, **File Format Notation** for a description of this notation.

Default Behaviour: When this section is listed as ''None'', it means that no files are created or modified as a consequence of direct action on the part of the utility when the utility is used as described by this document. However, the utility may create or modify system files, such as log files, that are outside the utility's normal execution environment.

EXTENDED DESCRIPTION

The **EXTENDED DESCRIPTION** section provides a place for describing the actions of very complicated utilities, such as text editors or language processors, which typically have elaborate command languages.

Default Behaviour: When this section is listed as ''None'', no further description is necessary.

EXIT STATUS

The **EXIT STATUS** section describes the values the utility will return to the calling program, or shell, and the conditions that cause these values to be returned. Usually, utilities return zero for successful completion and values greater than zero for various error conditions. If specific numeric values are listed in this section, the system will use those values for the errors described. In some cases, status values are listed more loosely, such as ''>0''. A portable application cannot rely on any specific value in the range shown and must be prepared to receive any value in the range.

For example, a utility may list zero as a successful return, 1 as a failure for a specific reason, and >1 as ''an error occurred''. In this case, unspecified conditions may cause a 2 or 3, or other value, to be returned. A portable application should be written so that it tests for successful exit status values (zero in this case), rather than relying upon the single specific error value listed in this document. In that way, it will have maximum portability, even on implementations with extensions.

Unspecified error conditions may be represented by specific values not listed in this document.

CONSEQUENCES OF ERRORS

The **CONSEQUENCES OF ERRORS** section describes the effects on the environment, file systems, process state, and so on, when error conditions occur. It does not describe error messages produced or exit status values used.

The many reasons for failure of a utility are generally not specified by the utility descriptions. Utilities may terminate prematurely if they encounter: invalid usage of options, arguments or environment variables; invalid usage of the complex syntaxes expressed in **EXTENDED DESCRIPTION** sections; difficulties accessing, creating, reading or writing files; or difficulties associated with the privileges of the process.

The following apply to each utility, unless otherwise stated:

- If the requested action cannot be performed on an operand representing a file, directory, user, process, etc., the utility will issue a diagnostic message to standard error and continue processing the next operand in sequence, but the final exit status is returned as non-zero.

  For a utility that recursively traverses a file hierarchy (such as *find* or *chown* **–R**), if the requested action cannot be performed on a file or directory encountered in the hierarchy, the utility will issue a diagnostic message to standard error and continue processing the remaining files in the hierarchy, but the final exit status will be returned as non-zero.

- If the requested action characterised by an option or option-argument cannot be performed, the utility will issue a diagnostic message to standard error and the exit status returned will be non-zero.

- When an unrecoverable error condition is encountered, the utility will exit with a non-zero exit status.

- A diagnostic message will be written to standard error whenever an error condition occurs.

When a utility encounters an error condition several actions are possible, depending on the severity of the error and the state of the utility. Included in the possible actions of various utilities are: deletion of temporary or intermediate work files; deletion of incomplete files; validity checking of the file system or directory.

Default Behaviour: When this section is listed as ‘‘Default’’, it means that any changes to the environment are unspecified.

APPLICATION USAGE

The **APPLICATION USAGE** section gives advice to the application programmer or user about the way the utility should be used.

EXAMPLES

The **EXAMPLES** section gives one or more examples of usage, where appropriate.

In all examples, quoting has been used, showing how sample commands (utility names combined with arguments) could be passed correctly to a shell (see *sh*) or as a string to the **XSH** specification *system*( ) function. Such quoting would not be used if the utility is invoked using one of the **XSH** specification *exec* functions.

FUTURE DIRECTIONS

The **FUTURE DIRECTIONS** section should be used as a guide to current thinking; there is not necessarily a commitment to implement all of these future directions in their entirety.

SEE ALSO

The **SEE ALSO** section lists related entries.

CHANGE HISTORY

The **CHANGE HISTORY** section shows the derivation of the description used by the

XSI and lists the functional differences between Issues 2, 3 and 4. Detailed listings of updates performed to align with the **XPG4** document are not given. For guidance on modifying applications that conform to Issue 3 to make them strictly compliant with this document, refer to the **Migration Guide**.

Certain of the standard utilities describe how they can invoke other utilities or applications, such as by passing a command string to the command interpreter. The external influences (**STDIN**, **ENVIRONMENT VARIABLES**, and so on) and external effects (**STDOUT**, **CONSEQUENCES OF ERRORS**, and so on) of such invoked utilities are not described in the section concerning the standard utility that invokes them.

Chapter 2

Shell Command Language

This chapter contains the definition of the XSI Shell Command Language.

2.1 Shell Introduction

The shell is a command language interpreter. This chapter describes the syntax of that command language as it is used by the *sh* utility and the *system*( ) and *popen*( ) functions in the **XSH** specification.

The shell operates according to the following general overview of operations. The specific details are included in the cited sections of this chapter.

1. The shell reads its input from a file (see *sh*), from the **−c** option or from the *system*( ) and *popen*( ) functions in the **XSH** specification. If the first line of a file of shell commands starts with the characters #!, the results are unspecified.

   The construct #! is reserved for implementations wishing to provide that extension. A portable application cannot use #! as the first line of a shell script; it might not be interpreted as a comment.

2. The shell breaks the input into tokens: words and operators. (See Section 2.3 on page 23.)

3. The shell parses the input into simple commands (see Section 2.9.1 on page 45) and compound commands (see Section 2.9.4 on page 52).

4. The shell performs various expansions (separately) on different parts of each command, resulting in a list of pathnames and fields to be treated as a command and arguments (see Section 2.6 on page 31).

5. The shell performs redirection (see Section 2.7 on page 40) and removes redirection operators and their operands from the parameter list.

6. The shell executes a function (see Section 2.9.5 on page 54), built-in (see Section 2.14 on page 67), executable file or script, giving the names of the arguments as positional parameters numbered 1 to *n*, and the name of the command (or in the case of a function within a script, the name of the script) as the positional parameter numbered 0 (see **Command Search and Execution** on page 47).

7. The shell optionally waits for the command to complete and collects the exit status (see Section 2.8.2 on page 44).

2.2 Quoting

Quoting is used to remove the special meaning of certain characters or words to the shell. Quoting can be used to preserve the literal meaning of the special characters in the next paragraph; prevent reserved words from being recognised as such; and prevent parameter expansion and command substitution within here-document processing (see Section 2.7.4 on page 41).

The following characters must be quoted if they are to represent themselves:

```
|  &  ;  <  >  (  )  $  ‘  \  "  ’  <space>  <tab>  <newline>
```

and the following may need to be quoted under certain circumstances. That is, these characters may be special depending on conditions described elsewhere in this document:

```
*   ?   [   #   ~   =   %
```

The various quoting mechanisms are the escape character, single-quotes and double-quotes. The here-document represents another form of quoting; see Section 2.7.4 on page 41.

2.2.1 Escape Character (Backslash)

A backslash that is not quoted preserves the literal value of the following character, with the exception of a newline character. If a newline character follows the backslash, the shell will interpret this as line continuation. The backslash and newline characters will be removed before splitting the input into tokens. Since the escaped newline character is removed entirely from the input and is not replaced by any white space, it cannot serve as a token separator.

2.2.2 Single-quotes

Enclosing characters in single-quotes (’ ’) preserves the literal value of each character within the single-quotes. A single-quote cannot occur within single-quotes.

A backslash cannot be used to escape a single-quote in a single-quoted string. An embedded quote can be created by writing, for example: ’a’\’’b’, which yields a’b. (See Section 2.6.5 on page 38 for a better understanding of how portions of words are either split into fields or remain concatenated.) A single token can be made up of concatenated partial strings containing all three kinds of quoting or escaping, thus permitting any combination of characters.

2.2.3 Double-quotes

Enclosing characters in double-quotes (" ") preserves the literal value of all characters within the double-quotes, with the exception of the characters dollar-sign, backquote and backslash, as follows:

$ The dollar-sign retains its special meaning introducing parameter expansion (see Section 2.6.2 on page 33), a form of command substitution (see Section 2.6.3 on page 36), and arithmetic expansion (see Section 2.6.4 on page 38).

  The input characters within the quoted string that are also enclosed between $(and the matching) will not be affected by the double-quotes, but rather define that command whose output replaces the $(...) when the word is expanded. The tokenising rules in Section 2.3 on page 23 are applied recursively to find the matching).

  Within the string of characters from an enclosed ${ to the matching }, an even number of unescaped double-quotes or single-quotes, if any, must occur. A preceding backslash character must be used to escape a literal { or }. The rule in Section 2.6.2 on page 33 will be used to determine the matching }.

‘ The backquote retains its special meaning introducing the other form of command substitution (see Section 2.6.3 on page 36). The portion of the quoted string from the initial backquote and the characters up to the next backquote that is not preceded by a backslash, having escape characters removed, defines that command whose output replaces ‘...‘ when the word is expanded. Either of the following cases produces undefined results:

- a single- or double-quoted string that begins, but does not end, within the ‘...‘ sequence
- a ‘...‘ sequence that begins, but does not end, within the same double-quoted string.

\ The backslash retains its special meaning as an escape character (see Section 2.2.1 on page 20) only when followed by one of the characters:

```
$      ‘      "      \      <newline>
```

A double-quote must be preceded by a backslash to be included within double-quotes. The parameter @ has special meaning inside double-quotes and is described in Section 2.5.2 on page 27.

In double-quoting, if a backslash is immediately followed by a character that would be interpreted as having a special meaning, the backslash is deleted and the subsequent character is taken literally. If a backslash does not precede a character that would have a special meaning, it is left in place unmodified and the character immediately following it is also left unmodified. Thus, for example:

```
"\$"  →  $
"\a"  →  \a
```

The requirement that double-quotes be matched inside ${...} within double-quotes and the rule for finding the matching } in Section 2.6.2 on page 33 eliminate several subtle inconsistencies in expansion for historical shells in rare cases; for example:

```
"${foo-bar"}
```

yields **bar** when **foo** is not defined, and is an invalid substitution when **foo** is defined, in many historical shells. The differences in processing the "${...}" form have led to inconsistencies between historical systems. A consequence of this rule is that single-quotes cannot be used to quote the } within "${...}"; for example:

```
unset bar
foo="${bar-’}’}"
```

is invalid because the "${...}" substitution contains an unpaired unescaped single-quote. The backslash can be used to escape the } in this example to achieve the desired result:

```
unset bar
foo="${bar-\}}"
```

Some systems have allowed the end of the word to terminate the backquoted command substitution, such as in:

```
"‘echo hello"
```

This usage is undefined; the matching backquote is required by this document. The other undefined usage can be illustrated by the example:

```
sh -c ’‘ echo "foo‘’
```

The description of the recursive actions involving command substitution can be illustrated with an example. Upon recognising the introduction of command substitution, the shell must parse input (in a new context), gathering the source for the command substitution until an unbalanced) or ‘ is located. For example, in the following:

```
echo "$(date; echo "
      one" )"
```

the double-quote following the *echo* does not terminate the first double-quote; it is part of the command substitution script. Similarly, in:

```
echo "$(echo *)"
```

the asterisk is not quoted since it is inside command substitution; however:

```
echo "$(echo "*")"
```

is quoted (and represents the asterisk character itself).

2.3 Token Recognition

The shell reads its input in terms of lines from a file, from a terminal in the case of an interactive shell or from a string in the case of *sh* **–c** or *system*( ). The input lines can be of unlimited length. These lines are parsed using two major modes: ordinary token recognition and processing of here-documents.

When an **io_here** token has been recognised by the grammar (see Section 2.10 on page 56), one or more of the subsequent lines immediately following the next **NEWLINE** token form the body of one or more here-documents and are parsed according to the rules of Section 2.7.4 on page 41.

When it is not processing an **io_here**, the shell will break its input into tokens by applying the first applicable rule below to the next character in its input. The token will be from the current position in the input until a token is delimited according to one of the rules below; the characters forming the token are exactly those in the input, including any quoting characters. If it is indicated that a token is delimited, and no characters have been included in a token, processing will continue until an actual token is delimited.

1. If the end of input is recognised, the current token will be delimited. If there is no current token, the end-of-input indicator will be returned as the token.

2. If the previous character was used as part of an operator and the current character is not quoted and can be used with the current characters to form an operator, it will be used as part of that (operator) token.

   Note that certain combinations of characters are invalid in portable scripts, as shown in the grammar, and that some systems have assigned these combinations (such as |&) as valid control operators. Portable scripts cannot rely on receiving errors in all cases where this document indicates that a syntax is invalid.

3. If the previous character was used as part of an operator and the current character cannot be used with the current characters to form an operator, the operator containing the previous character will be delimited.

4. If the current character is backslash, single-quote or double-quote (\, ’ or ") and it is not quoted, it will affect quoting for subsequent characters up to the end of the quoted text. The rules for quoting are as described in Section 2.2 on page 20. During token recognition no substitutions will be actually performed, and the result token will contain exactly the characters that appear in the input (except for newline character joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the quote mark and the end of the quoted text. The token will not be delimited by the end of the quoted field.

5. If the current character is an unquoted $ or ‘, the shell will identify the start of any candidates for parameter expansion (Section 2.6.2 on page 33), command substitution (Section 2.6.3 on page 36), or arithmetic expansion (Section 2.6.4 on page 38) from their introductory unquoted character sequences: $ or ${, $(or ‘, and $((, respectively. The shell will read sufficient input to determine the end of the unit to be expanded (as explained in the cited sections). While processing the characters, if instances of expansions or quoting are found nested within the substitution, the shell will recursively process them in the manner specified for the construct that is found. The characters found from the beginning of the substitution to its end, allowing for any recursion necessary to recognise embedded constructs, will be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token will not be delimited by the end of the substitution.

6. If the current character is not quoted and can be used as the first character of a new operator, the current token (if any) will be delimited. The current character will be used as the beginning of the next (operator) token.
7. If the current character is an unquoted newline character, the current token will be delimited.
8. If the current character is an unquoted blank character, any token containing the previous character is delimited and the current character will be discarded.
9. If the previous character was part of a word, the current character will be appended to that word.
10. If the current character is a #, it and all subsequent characters up to, but excluding, the next newline character will be discarded as a comment. The newline character that ends the line is not considered part of the comment. The # starts a comment only when it is at the beginning of a token. Since the search for the end-of-comment does not consider an escaped newline character specially, a comment cannot be continued to the next line.
11. The current character will be used as the start of a new word.

Once a token is delimited, it will be categorised as required by the grammar in Section 2.10 on page 56.

2.3.1 Alias Substitution

The processing of aliases is supported on all X/Open systems.

After a token has been delimited, but before applying the grammatical rules in Section 2.10 on page 56, a resulting word that is identified to be the command name word of a simple command is examined to determine if it is an unquoted, valid alias name. However, reserved words in correct grammatical context are not candidates for alias substitution. A valid alias name (see the term alias name in the **XBD** specification, **Chapter 2**, **Glossary**) is one that has been defined by the *alias* utility and not subsequently undefined using *unalias*. Implementations also may provide predefined valid aliases that are in effect when the shell is invoked. To prevent infinite loops in recursive aliasing, if the shell is not currently processing an alias of the same name, the word will be replaced by the value of the alias; otherwise, it will not be replaced.

If the value of the alias replacing the word ends in a blank character, the shell will check the next command word for alias substitution; this process continues until a word is found that is not a valid alias or an alias value does not end in a blank character.

When used as specified by this document, alias definitions will not be inherited by separate invocations of the shell or by the utility execution environments invoked by the shell; see Section 2.12 on page 63.

The definition of *alias name* precludes an alias name containing a slash character. Since the text applies to the command words of simple commands, reserved words (in their proper places) cannot be confused with aliases.

An example concerning trailing blank characters and reserved words follows. If the user types:

```
$ alias foo="/bin/ls "
$ alias while="/"
```

The effect of executing:

```
$ while true
> do
> echo "Hello, World"
> done
```

is a never-ending sequence of **Hello, World** strings to the screen. However, if the user types:

```
$ foo while
```

the result will be an *ls* listing of **/**. Since the alias substitution for **foo** ends in a space character, the next word is checked for alias substitution. The next word, **while**, has also been aliased, so it is substituted as well. Since it is not in the proper position as a command word, it is not recognised as a reserved word.

If the user types:

```
$ foo; while
```

while retains its normal reserved-word properties.

2.4 Reserved Words

Reserved words are words that have special meaning to the shell. (See Section 2.9 on page 45.) The following words will be recognised as reserved words:

!	**elif**	**fi**	**in**	**while**
case	**else**	**for**	**then**	**{***
do	**esac**	**if**	**until**	**}**
done				

This recognition will occur only when none of the characters are quoted and when the word is used as:

- the first word of a command
- the first word following one of the reserved words other than **case**, **for**, or **in**
- the third word in a **case** or **for** command (only **in** is valid in this case).

See the grammar in Section 2.10 on page 56.

The following words may be recognised as reserved words on some systems (when none of the characters are quoted), causing unspecified results:

function **select** **[[** **]]**

This list of unspecified reserved words is from the KornShell, so portable applications cannot use them in places a reserved word would be recognised without quoting some or all of their characters.

Words that are the concatenation of a name and a colon (:) are reserved; their use produces unspecified results. This reservation is to allow future implementations that support named labels for flow control.

Reserved words are recognised only when they are delimited (that is, meet the definition of **word** in the **XSH** specification), whereas operators are themselves delimiters. For instance, (and) are control operators, so that no space character is needed in (list). However, { and } are reserved words in { list;}, so that in this case the leading space character and semicolon are required.

---

* In some historical systems, the curly braces are treated as control operators. To assist in future standardisation activities, portable applications should avoid using unquoted braces to represent the characters themselves. It is possible that a future version of the ISO/IEC 9945-2: 1993 standard may require that { and } be treated individually as control operators, although the token {} will probably be a special-case exemption from this because of the often-used *find*{} construct.

2.5 Parameters and Variables

A parameter can be denoted by a name, a number or one of the special characters listed in Section 2.5.2. A variable is a parameter denoted by a name.

A parameter is set if it has an assigned value (null is a valid value). Once a variable is set, it can only be unset by using the *unset* special built-in command.

2.5.1 Positional Parameters

A positional parameter is a parameter denoted by the decimal value represented by one or more digits, other than the single digit 0. The digits denoting the positional parameters are always interpreted as a decimal value, even if there is a leading zero. When a positional parameter with more than one digit is specified, the application must enclose the digits in braces (see Section 2.6.2 on page 33). Positional parameters are initially assigned when the shell is invoked (see *sh*), temporarily replaced when a shell function is invoked (see Section 2.9.5 on page 54), and can be reassigned with the *set* special built-in command.

2.5.2 Special Parameters

Listed below are the special parameters and the values to which they will expand. Only the values of the special parameters are listed; see Section 2.6 on page 31 for a detailed summary of all the stages involved in expanding words.

***** Expands to the positional parameters, starting from one. When the expansion occurs within a double-quoted string (see Section 2.2.3 on page 20), it expands to a single field with the value of each parameter separated by the first character of the *IFS* variable, or by a space character if *IFS* is unset. If *IFS* is set to a null string, this is not equivalent to unsetting it; its first character will not exist, so the parameter values are concatenated. For example:

```
$ IFS=''
$ set foo bar bam
$ echo "$@"
foo bar bam
$ echo "$*"
foobarbam
$ unset IFS
$ echo "$*"
foo bar bam
```

@ Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, and where field splitting (see Section 2.6.5 on page 38) is performed, each positional parameter expands as a separate field, with the provision that the expansion of the first parameter is still joined with the beginning part of the original word (assuming that the expanded parameter was embedded within a word), and the expansion of the last parameter is still joined with the last part of the original word. If there are no positional parameters, the expansion of @ generates zero fields, even when @ is double-quoted.

\# Expands to the decimal number of positional parameters. The command name (parameter 0) is not counted in the number given by # because it is a special parameter, not a positional parameter.

? Expands to the decimal exit status of the most recent pipeline (see Section 2.9.2 on page 49).

– (Hyphen.) Expands to the current option flags (the single-letter option names concatenated into a string) as specified on invocation, by the *set* special built-in command or implicitly by the shell.

The $– special parameter can be used to save and restore *set* options:

```
Save=$(echo $- | sed 's/[ics]//g')
...
set +aCefnuvx
if [ -n "$Save" ]; then
       set -$Save
fi
```

The three options are removed using *sed* in the example because they may appear in the value of $– (from the *sh* command line), but are not valid options to *set*.

$ Expands to the decimal process ID of the invoked shell. In a subshell (see Section 2.12 on page 63), $ expands to the same value as that of the current shell.

Most historical implementations implement subshells by forking; thus, the special parameter $ does not necessarily represent the process ID of the shell process executing the commands since the subshell execution environment preserves the value of $.

! Expands to the decimal process ID of the most recent background command (see Section 2.9.3 on page 49) executed from the current shell. (For example, background commands executed from subshells do not affect the value of $! in the current shell environment.) For a pipeline, the process ID is that of the last command in the pipeline.

0 (Zero.) Expands to the name of the shell or shell script. See *sh* for a detailed description of how this name is derived.

See the description of the *IFS* variable in Section 2.5.3 on page 29.

The descriptions of parameters * and @ assume the reader is familiar with the field splitting discussion in Section 2.6.5 on page 38 and understands that portions of the word will remain concatenated unless there is some reason to split them into separate fields.

Some examples of the * and @ properties, including the concatenation aspects:

```
set "abc" "def ghi" "jkl"

echo $*       => "abc" "def" "ghi" "jkl"
echo "$*"     => "abc def ghi jkl"
echo $@       => "abc" "def" "ghi" "jkl"
```

but

```
echo "$@"     => "abc" "def ghi" "jkl"
echo "xx$@yy"=> "xxabc" "def ghi" "jklyy"
echo "$@$@"  => "abc" "def ghi" "jklabc" "def ghi" "jkl"
```

In the preceding examples, the double-quote characters that appear after the => do not appear in the output and are used only to illustrate word boundaries.

2.5.3 Shell Variables

Variables are initialised from the environment (as defined by the **XSH** specification) and can be given new values with variable assignment commands. If a variable is initialised from the environment, it is marked for export immediately; see the *export* special built-in. New variables can be defined and initialised with variable assignments, with the *read* or *getopts* utilities, with the *name* parameter in a **for** loop, with the ${*name*=*word*} expansion or with other mechanisms provided as implementation extensions. The following variables affect the execution of the shell:

ENV This variable, when the shell is invoked, is subjected to parameter expansion (see Section 2.6.2 on page 33) by the shell and the resulting value is used as a pathname of a file containing shell commands to execute in the current environment. The file need not be executable. If the expanded value of *ENV* is not an absolute pathname, the results are unspecified. *ENV* will be ignored if the user's real and effective user IDs or real and effective group IDs are different.

This variable can be used to set aliases and other items local to the invocation of a shell. The file referred to by *ENV* differs from **$HOME/.profile** in that **.profile** is typically executed at session startup, whereas the *ENV* file is executed at the beginning of each shell invocation. The *ENV* value is interpreted in a manner similar to a dot script, in that the commands are executed in the current environment and the file needs to be readable, but not executable. However, unlike dot scripts, no *PATH* searching is performed. This is used as a guard against Trojan Horse security breaches.

HOME This variable is interpreted as the pathname of the user's home directory. The contents of *HOME* are used in tilde expansion (see Section 2.6.1 on page 32).

IFS *Input field separators*: a string treated as a list of characters that is used for field splitting and to split lines into fields with the *read* command. If *IFS* is not set, the shell will behave as if the value of *IFS* were the space, tab and newline characters. (See Section 2.6.5 on page 38.)

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL This variable provides a default value for the *LC_** variables, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

LC_COLLATE This variable determines the behaviour of range expressions, equivalence classes and multi-character collating elements within pattern matching.

LC_CTYPE This variable determines the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters), which characters are defined as letters (character class **alpha**) and blank characters (character class **blank**), and the behaviour of character classes within pattern matching. Changing the value of *LC_CTYPE* after the shell has started does not affect the lexical processing of shell commands in the current shell execution environment or its subshells. Invoking a shell script or performing *exec sh* subjects the new shell to the changes in *LC_CTYPE*.

LC_MESSAGES This variable determines the language in which messages should be written.

LINENO This variable is set by the shell to a decimal number representing the current sequential line number (numbered starting with 1) within a script or function before it executes each command. If the user unsets or resets *LINENO*, the variable may lose its special meaning for the life of the shell. If the shell is not currently executing a script or function, the value of *LINENO* is unspecified.

EX *NLSPATH* Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH This variable represents a string formatted as described in the **XBD** specification, **Chapter 6**, **Environment Variables**, used to effect command interpretation. See **Command Search and Execution** on page 47.

PPID This variable is set by the shell to the decimal process ID of the process that invoked this shell. In a subshell (see Section 2.12 on page 63), *PPID* will be set to the same value as that of the parent of the current shell. For example, *echo* $PPID and (*echo* $PPID) would produce the same value.

Without this variable, there is no way for a utility to signal its parent or to find its parent process. This is also useful to know if the shell has been orphaned.

PS1 Each time an interactive shell is ready to read a command, the value of this variable is subjected to parameter expansion and written to standard error. The default value is "$ ". For users who have specific additional implementation-dependent privileges, the default may be another, implementation-dependent, value. (Historically, the superuser has had a prompt of "# ".) The shell replaces each instance of the character ! in *PS1* with the history file number of the next command to be typed. Escaping the ! with another ! (that is, !!) places the literal character ! in the prompt.

PS2 Each time the user enters a newline character prior to completing a command line in an interactive shell, the value of this variable is subjected to parameter expansion and written to standard error. The default value is "> ".

PS4 When an execution trace (*set* **–x**) is being performed in an interactive shell, before each line in the execution trace, the value of this variable is subjected to parameter expansion and written to standard error. The default value is "+ ".

For example, the following script:

```
PS4='[${LINENO}]+ '
set –x
echo Hello
```

writes the following to standard error:

```
[3]+ echo Hello
```

Tilde expansion for components of the *PATH* in an assignment such as:

```
PATH=~hlj/bin:~dwc/bin:$PATH
```

is a feature of some historical shells and is allowed by the wording of Section 2.6.1 on page 32. Note that the tildes are expanded during the assignment to *PATH*, not when *PATH* is accessed during command search.

2.6 Word Expansions

This section describes the various expansions that are performed on words. Not all expansions are performed on every word, as explained in the following sections.

Tilde expansions, parameter expansions, command substitutions, arithmetic expansions and quote removals that occur within a single word expand to a single field. It is only field splitting or pathname expansion that can create multiple fields from a single word. The single exception to this rule is the expansion of the special parameter @ within double-quotes, as described in Section 2.5.2 on page 27.

The order of word expansion is as follows:

1. Tilde expansion (see Section 2.6.1 on page 32), parameter expansion (see Section 2.6.2 on page 33), command substitution (see Section 2.6.3 on page 36), and arithmetic expansion (see Section 2.6.4 on page 38) are performed, beginning to end. See item 5 in Section 2.3 on page 23.
2. Field splitting (see Section 2.6.5 on page 38) is performed on the portions of the fields generated by step 1, unless *IFS* is null.
3. Pathname expansion (see Section 2.6.6 on page 39) is performed, unless *set* –**f** is in effect.
4. Quote removal (see Section 2.6.7 on page 39) always is performed last.

The expansions described in this section will occur in the same shell environment as that in which the command is executed.

If the complete expansion appropriate for a word results in an empty field, that empty field will be deleted from the list of fields that form the completely expanded command, unless the original word contained single-quote or double-quote characters.

The $ character is used to introduce parameter expansion, command substitution or arithmetic evaluation. If an unquoted $ is followed by a character that is either not numeric, the name of one of the special parameters (see Section 2.5.2 on page 27), a valid first character of a variable name, a left curly brace ({) or a left parenthesis, the result is unspecified.

IFS is used for performing field splitting on the results of parameter and command substitution; it is not used for splitting all fields. Previous versions of the shell used it for splitting all fields during field splitting, but this has severe problems because the shell can no longer parse its own script. There are also important security implications caused by this behaviour. All useful applications of *IFS* use it for parsing input of the *read* utility and for splitting the results of parameter and command substitution.

The rule concerning expansion to a single field requires that if **foo=abc** and **bar=def**, that:

```
"$foo""$bar"
```

expands to the single field:

```
abcdef
```

The rule concerning empty fields can be illustrated by:

```
$  unset foo
$  set $foo bar '' xyz "$foo" abc
$  for i
>  do
>          echo "-$i-"
>  done
-bar-
--
-xyz-
--
-abc-
```

Step 2 indicates that parameter expansion, command substitution and arithmetic expansion are all processed simultaneously as they are scanned. For example, the following is valid arithmetic:

```
x=1
echo $(( $(echo 3)+$x ))
```

2.6.1 Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, followed by all of the characters preceding the first unquoted slash in the word, or all the characters in the word if there is no slash. In an assignment (see **variable assignment** in the **XBD** specification, **Chapter 2**, **Glossary**) multiple tilde-prefixes can be used: at the beginning of the word (that is, following the equal sign of the assignment), following any unquoted colon or both. A tilde-prefix in an assignment is terminated by the first unquoted colon or slash. If none of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix following the tilde are treated as a possible login name from the user database. A portable login name cannot contain characters outside the set given in the description of the *LOGNAME* environment variable in the **XSH** specification. If the login name is null (that is, the tilde-prefix contains only the tilde), the tilde-prefix will be replaced by the value of the variable *HOME*. If *HOME* is unset, the results are unspecified. Otherwise, the tilde-prefix will be replaced by a pathname of the home directory associated with the login name obtained using the **XSH** specification *getpwnam*( ) function. If the system does not recognise the login name, the results are undefined.

Tilde expansion generally occurs only at the beginning of words, but an exception based on historical practice has been included:

```
PATH=/posix/bin:~dgk/bin
```

is eligible for tilde expansion because tilde follows a colon and none of the relevant characters is quoted. Consideration was given to prohibiting this behaviour because any of the following are reasonable substitutes:

OB
```
PATH=$(printf %s: ~rms/bin ~bfox/bin ...)
```

```
PATH=$(printf %s ~karels/bin : ~bostic/bin)
```

```
for Dir in ~maart/bin ~srb/bin ...
do
       PATH=${PATH:+$PATH:}$Dir
done
```

OB In the first command, any number of directory names are concatenated and separated with colons, but it may be undesirable to end the variable with a colon because this is an obsolescent means to include dot at the end of the *PATH*. In the second, explicit colons are used for each

directory. In all cases, the shell performs tilde expansion on each directory because all are separate words to the shell.

Note that expressions in operands such as:

```
make -k mumble LIBDIR=~chet/lib
```

do not qualify as shell variable assignments and tilde expansion is not performed (unless the command does so itself, which *make* does not).

The special sequence **$~** has been designated for future implementations to evaluate as a means of forcing tilde expansion in any word.

Because of the requirement that the word is not quoted, the following are not equivalent; only the last will cause tilde expansion:

```
\~hlj/   ~h\lj/   ~"hlj"/   ~hlj\/   ~hlj/
```

The results of giving tilde with an unknown login name are undefined because the KornShell ~+ and ~– constructs make use of this condition, but, in general it is an error to give an incorrect login name with tilde. The results of having *HOME* unset are unspecified because some historical shells treat this as an error.

2.6.2 Parameter Expansion

The format for parameter expansion is as follows:

```
${expression}
```

where *expression* consists of all characters until the matching }. Any } escaped by a backslash or within a quoted string, and characters in embedded arithmetic expansions, command substitutions and variable expansions, are not examined in determining the matching }.

The simplest form for parameter expansion is:

```
${parameter}
```

The value, if any, of *parameter* will be substituted.

The parameter name or symbol can be enclosed in braces, which are optional except for positional parameters with more than one digit or when *parameter* is followed by a character that could be interpreted as part of the name. The matching closing brace will be determined by counting brace levels, skipping over enclosed quoted strings and command substitutions.

If the parameter name or symbol is not enclosed in braces, the expansion will use the longest valid name (see **name** in the **XBD** specification, **Chapter 2**, **Glossary**), whether or not the symbol represented by that name exists. When the shell is scanning its input to determine the boundaries of a name, it is not bound by its knowledge of what names are already defined. For example, if **F** is a defined shell variable, the command:

```
echo $Fred
```

does not echo the value of **$F** followed by **red**; it selects the longest possible valid name, **Fred**, which in this case might be unset.

If a parameter expansion occurs inside double-quotes:

- Pathname expansion will not be performed on the results of the expansion.
- Field splitting will not be performed on the results of the expansion, with the exception of @; see Section 2.5.2 on page 27.

In addition, a parameter expansion can be modified by using one of the following formats. In each case that a value of *word* is needed (based on the state of *parameter*, as described below), *word* will be subjected to tilde expansion, parameter expansion, command substitution and arithmetic expansion. If *word* is not needed, it will not be expanded. The } character that delimits the following parameter expansion modifications is determined as described previously in this section and in Section 2.2.3 on page 20. (For example, ${foo-bar}xyz} would result in the expansion of **foo** followed by the string xyz} if **foo** is set, else the string barxyz}).

${*parameter*:–*word*} **Use Default Values.** If *parameter* is unset or null, the expansion of *word* will be substituted; otherwise, the value of *parameter* will be substituted.

${*parameter*:=*word*} **Assign Default Values.** If *parameter* is unset or null, the expansion of *word* will be assigned to *parameter*. In all cases, the final value of *parameter* will be substituted. Only variables, not positional parameters or special parameters, can be assigned in this way.

${*parameter*:?**[***word***]**} **Indicate Error if Null or Unset.** If *parameter* is unset or null, the expansion of *word* (or a message indicating it is unset if *word* is omitted) will be written to standard error and the shell will exit with a non-zero exit status. Otherwise, the value of *parameter* will be substituted. An interactive shell need not exit.

${*parameter*:+*word*} **Use Alternative Value.** If *parameter* is unset or null, null will be substituted; otherwise, the expansion of *word* will be substituted.

In the parameter expansions shown previously, use of the colon in the format results in a test for a parameter that is unset or null; omission of the colon results in a test for a parameter that is only unset. The following table summarises the effect of the colon:

	parameter **set and not null**	***parameter*** **set but null**	***parameter*** **unset**
${*parameter***:-***word***}**	substitute *parameter*	substitute *word*	substitute *word*
${*parameter***–***word***}**	substitute *parameter*	substitute null	substitute *word*
${*parameter***:=***word***}**	substitute *parameter*	assign *word*	assign *word*
${*parameter***=***word***}**	substitute *parameter*	substitute *parameter*	assign null
${*parameter***:?***word***}**	substitute *parameter*	error, exit	error, exit
${*parameter***?***word***}**	substitute *parameter*	substitute null	error, exit
${*parameter***:+***word***}**	substitute *word*	substitute null	substitute null
${*parameter***+***word***}**	substitute *word*	substitute *word*	substitute null

In all cases shown with ‘‘substitute’’, the expression is replaced with the value shown. In all cases shown with ‘‘assign’’, *parameter* is assigned that value, which also replaces the expression.

${#*parameter***}** **String Length**. The length in characters of the value of *parameter*. If *parameter* is * or @, the result of the expansion is unspecified.

The following four varieties of parameter expansion provide for substring processing. In each case, pattern matching notation (see Section 2.13 on page 64), rather than regular expression notation, will be used to evaluate the patterns. If *parameter* is * or @, the result of the expansion is unspecified. Enclosing the full parameter expansion string in double-quotes will not cause the following four varieties of pattern characters to be quoted, whereas quoting characters within the braces will have this effect.

${*parameter*%*word*} **Remove Smallest Suffix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the smallest portion of the suffix matched by the *pattern* deleted.

${*parameter*%%*word*} **Remove Largest Suffix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the largest portion of the suffix matched by the *pattern* deleted.

${*parameter*#*word*} **Remove Smallest Prefix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the smallest portion of the prefix matched by the *pattern* deleted.

${*parameter*##*word*} **Remove Largest Prefix Pattern**. The *word* will be expanded to produce a pattern. The parameter expansion then will result in *parameter*, with the largest portion of the prefix matched by the *pattern* deleted.

Examples

${*parameter*:–*word*}

In this example, *ls* is executed only if x is null or unset. (The $(ls) command substitution notation is explained in Section 2.6.3 on page 36.)

```
${x:-$(ls)}
```

${*parameter*:=*word*}

```
unset X
echo ${X:=abc}
abc
```

${*parameter*:?*word*}

```
unset posix
echo ${posix:?}
sh: posix: parameter null or not set
```

${*parameter*:+*word*}

```
set a b c
echo ${3:+posix}
posix
```

${#*parameter***}**

```
HOME=/usr/posix
echo ${#HOME}
10
```

${*parameter%word*}

```
x=file.c
echo ${x%.c}.o
file.o
```

${*parameter%%word*}

```
x=posix/src/std
echo ${x%%/*}
posix
```

${*parameter#word*}

```
x=$HOME/src/cmd
echo ${x#$HOME}
/src/cmd
```

${*parameter##word*}

```
x=/one/two/three
echo ${x##*/}
three
```

The double-quoting of patterns is different depending on where the double-quotes are placed:

"${x#*}" The asterisk is a pattern character.

${x#"*"} The literal asterisk is quoted and not special.

2.6.3 Command Substitution

Command substitution allows the output of a command to be substituted in place of the command name itself. Command substitution occurs when the command is enclosed as follows:

`$(`*command*`)`

or (backquoted version):

`` `command` ``

The shell will expand the command substitution by executing *command* in a subshell environment (see Section 2.12 on page 63) and replacing the command substitution (the text of *command* plus the enclosing $( ) or backquotes) with the standard output of the command, removing sequences of one or more newline characters at the end of the substitution. Embedded newline characters before the end of the output will not be removed; however, they may be treated as field delimiters and eliminated during field splitting, depending on the value of *IFS* and quoting that is in effect.

Within the backquoted style of command substitution, backslash shall retain its literal meaning, except when followed by:

`$ ` ` \`

(dollar-sign, backquote, backslash). The search for the matching backquote is satisfied by the first backquote found without a preceding backslash; during this search, if a non-escaped backquote is encountered within a shell comment, a here-document, an embedded command substitution of the $(*command*) form, or a quoted string, undefined results occur. A single- or double-quoted string that begins, but does not end, within the `` `...` `` sequence produces undefined results.

With the $(*command*) form, all characters following the open parenthesis to the matching closing parenthesis constitute the *command*. Any valid shell script can be used for *command*, except:

- A script consisting solely of redirections produces unspecified results.
- See the restriction on single subshells described below.

The results of command substitution will not be field splitting and pathname expansion processed for further tilde expansion, parameter expansion, command substitution or arithmetic expansion. If a command substitution occurs inside double-quotes, it will not be performed on the results of the substitution.

Command substitution can be nested. To specify nesting within the backquoted version, the application must precede the inner backquotes with backslashes; for example:

```
\‘command\‘
```

The $( ) form of command substitution solves a problem of inconsistent behaviour when using backquotes. For example:

Command	Output
`echo ’\$x’`	`\$x`
`echo ‘echo ’\$x’‘`	`$x`
`echo $(echo ’\$x’)`	`\$x`

Additionally, the backquoted syntax has historical restrictions on the contents of the embedded command. While the new $( ) form can process any kind of valid embedded script, the backquoted form cannot handle some valid scripts that include backquotes. For example, these otherwise valid embedded scripts do not work in the left column, but do work on the right:

```
echo ‘                          echo $(
cat <<\eof                      cat <<\eof
a here-doc with ‘               a here-doc with )
eof                             eof
‘                               )

echo ‘                          echo $(
echo abc # a comment with ‘     echo abc # a comment with )
‘                               )

echo ‘                          echo $(
echo ’’’                        echo ’)’
‘                               )
```

Because of these inconsistent behaviours, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts.

If the command substitution consists of a single subshell, such as:

```
$( (command) )
```

a portable application must separate the $(and (into two tokens (that is, separate them with white space). This is required to avoid any ambiguities with arithmetic expansion.

2.6.4 Arithmetic Expansion

Arithmetic expansion provides a mechanism for evaluating an arithmetic expression and substituting its value. The format for arithmetic expansion is as follows:

```
$((expression))
```

The expression is treated as if it were in double-quotes, except that a double-quote inside the expression is not treated specially. The shell will expand all tokens in the expression for parameter expansion, command substitution and quote removal.

Next, the shell will treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression will be processed according to the rules of the ISO C standard, with the following exceptions:

- Only integer arithmetic is required.
- The sizeof( ) operator and the prefix and postfix ++ and −− operators are not required.
- Selection, iteration and jump statements are not supported.

As an extension, the shell may recognise arithmetic expressions beyond those listed. If the expression is invalid, the expansion will fail and the shell will write a message to standard error indicating the failure.

A simple example using arithmetic expansion:

```
# repeat a command 100 times
x=100
while [ $x -gt 0 ]
do
       command
       x=$(($x-1))
done
```

2.6.5 Field Splitting

After parameter expansion (Section 2.6.2 on page 33), command substitution (Section 2.6.3 on page 36), and arithmetic expansion (Section 2.6.4) the shell will scan the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell will treat each character of the *IFS* as a delimiter and use the delimiters to split the results of parameter expansion and command substitution into fields.

1. If the value of *IFS* is a space, tab and newline character, or if it is unset, any sequence of space, tab or newline characters at the beginning or end of the input will be ignored and any sequence of those characters within the input will delimit a field. For example, the input:

   ```
   <newline><space><tab>foo<tab><tab>bar<space>
   ```

   yields two fields, **foo** and **bar**.

2. If the value of *IFS* is null, no field splitting will be performed.

3. Otherwise, the following rules will be applied in sequence. The term ‘‘*IFS* white space’’ is used to mean any sequence (zero or more instances) of white-space characters that are in the *IFS* value (for example, if *IFS* contains space/comma/tab, any sequence of space and tab characters is considered *IFS* white space).

a. *IFS* white space is ignored at the beginning and end of the input.

b. Each occurrence in the input of an *IFS* character that is not *IFS* white space, along with any adjacent *IFS* white space, will delimit a field, as described previously.

c. Non-zero-length *IFS* white space will delimit a field.

The last rule can be summarised as a pseudo-ERE:

```
(s*ns*|s+)
```

where *s* is an *IFS* white-space character and *n* is a character in the *IFS* that is not white space. Any string matching that ERE delimits a field, except that the *s*+ form does not delimit fields at the beginning or the end of a line. For example, if *IFS* is space/comma/tab, the string:

```
<space><space>red<space><space>,<space>white<space>blue
```

yields the three colours as the delimited fields.

2.6.6 Pathname Expansion

After field splitting, if *set* –**f** is not in effect, each field in the resulting command line will be expanded using the algorithm described in Section 2.13 on page 64, qualified by the rules in Section 2.13.3 on page 66.

2.6.7 Quote Removal

The quote characters:

```
\       '       "
```

(backslash, single-quote, double-quote) that were present in the original word will be removed unless they have themselves been quoted.

2.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see Section 2.12 on page 63) or for any command. *Redirection operators* can be used with numbers representing file descriptors (see the definition in the ISO POSIX-1 standard) as described below.

The overall format used for redirection is:

```
[n]redir-op  word
```

The number *n* is an optional decimal number designating the file descriptor number; it must be delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number will not be recognised as part of the redirection expression. For example:

```
echo \2>a
```

writes the character 2 into file **a**. If any part of *redir-op* is quoted, no redirection expression will be recognised. For example:

```
echo 2\>a
```

writes the characters 2>a to standard output. The optional number, redirection operator and *word* will not appear in the arguments provided to the command to be executed (if any).

Open files are represented by decimal numbers starting with zero. The largest possible value is implementation-dependent; however, all implementations support at least 0 to 9, inclusive, for use by the application. These numbers are called *file descriptors*. The values 0, 1 and 2 have special meaning and conventional uses and are implied by certain redirection operations; they are referred to as *standard input*, *standard output* and *standard error*, respectively. Programs usually take their input from standard input, and write output on standard output. Error messages are usually written on standard error. The redirection operators can be preceded by one or more digits (with no intervening blank characters allowed) to designate the file descriptor number.

If the redirection operator is << or <<–, the word that follows the redirection operator will be subjected to quote removal; it is unspecified whether any of the other expansions occur. For the other redirection operators, the word that follows the redirection operator will be subjected to tilde expansion, parameter expansion, command substitution, arithmetic expansion and quote removal. Pathname expansion will not be performed on the word by a non-interactive shell; an interactive shell may perform it, but will do so only when the expansion would result in one word.

If more than one redirection operator is specified with a command, the order of evaluation is from beginning to end.

A failure to open or create a file will cause the redirection to fail.

2.7.1 Redirecting Input

Input redirection will cause the file whose name results from the expansion of *word* to be opened for reading on the designated file descriptor, or standard input if the file descriptor is not specified.

The general format for redirecting input is:

```
[n]<word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection will refer to standard input (file descriptor 0).

2.7.2 Redirecting Output

The two general formats for redirecting output are:

```
[n]>word
[n]>|word
```

where the optional *n* represents the file descriptor number. If the number is omitted, the redirection will refer to standard output (file descriptor 1).

Output redirection using the > format will fail if the *noclobber* option is set (see the description of *set* –**C**) and the file named by the expansion of *word* exists and is a regular file. Otherwise, redirection using the > or >| formats will cause the file whose name results from the expansion of *word* to be created and opened for output on the designated file descriptor, or standard output if none is specified. If the file does not exist, it will be created; otherwise, it will be truncated to be an empty file after being opened.

2.7.3 Appending Redirected Output

Appended output redirection will cause the file whose name results from the expansion of word to be opened for output on the designated file descriptor. The file is opened as if the **XSH** specification *open*( ) function was called with the O_APPEND flag. If the file does not exist, it will be created.

The general format for appending redirected output is as follows:

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

where the optional *n* represents the file descriptor number.

2.7.4 Here-document

The redirection operators << and <<– both allow redirection of lines contained in a shell input file, known as a *here-document*, to the standard input of a command.

The here-document is treated as a single word that begins after the next newline character and continues until there is a line containing only the delimiter, with no trailing blank characters. Then the next here-document starts, if there is one. The format is as follows:

```
[n]<<word
        here-document
delimiter
```

If any character in *word* is quoted, the delimiter is formed by performing quote removal on *word*, and the here-document lines will not be expanded. Otherwise, the delimiter is the *word* itself.

If no characters in *word* are quoted, all lines of the here-document will be expanded for parameter expansion, command substitution and arithmetic expansion. In this case, the backslash in the input will behave as the backslash inside double-quotes (see Section 2.2.3 on page 20). However, the double-quote character (") will not be treated specially within a here-document, except when the double-quote appears within $( ), ` ` or ${ }.

If the redirection symbol is <<–, all leading tab characters will be stripped from input lines and the line containing the trailing delimiter. If more than one << or <<– operator is specified on a line, the here-document associated with the first operator will be supplied first by the application and will be read first by the shell. For example:

```
cat <<eof1; cat <<eof2
Hi,
eof1
Helene.
eof2
```

The case of a missing delimiter at the end of a here-document is not specified. This is considered an error in the script (one that sometimes can be difficult to diagnose), although some systems have treated end-of-file as an implicit delimiter.

2.7.5 Duplicating an Input File Descriptor

The redirection operator:

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

is used to duplicate one input file descriptor from another, or to close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard input if *n* is not specified, will be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for input, a redirection error will result (see Section 2.8.1 on page 44). If *word* evaluates to –, file descriptor *n*, or standard input if *n* is not specified, will be closed. If *word* evaluates to something else, the behaviour is unspecified.

2.7.6 Duplicating an Output File Descriptor

The redirection operator:

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

is used to duplicate one output file descriptor from another, or to close one. If *word* evaluates to one or more digits, the file descriptor denoted by *n*, or standard output if *n* is not specified, will be made to be a copy of the file descriptor denoted by *word*; if the digits in *word* do not represent a file descriptor already open for output, a redirection error will result (see Section 2.8.1 on page 44). If *word* evaluates to –, file descriptor *n*, or standard output if *n* is not specified, will be closed. If *word* evaluates to something else, the behaviour is unspecified.

The construct 2>&1 is often used to redirect standard error to the same file as standard output. Since the redirections take place beginning to end, the order of redirections is significant. For example:

```
ls > foo 2>&1
```

directs both standard output and standard error to file **foo**. However:

```
ls 2>&1 > foo
```

only directs standard output to file **foo** because standard error was duplicated as standard output before standard output was directed to file **foo**.

2.7.7 Open File Descriptors for Reading and Writing

The redirection operator:

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

will cause the file whose name is the expansion of *word* to be opened for both reading and writing on the file descriptor denoted by *n*, or standard input if *n* is not specified. If the file does not exist, it will be created.

The <> operator could be useful in writing an application that worked with several terminals, and occasionally wanted to start up a shell. That shell would in turn be unable to run applications that run from an ordinary controlling terminal unless it could make use of <> redirection. The specific example is a historical version of the pager *more*, which reads from standard error to get its commands, so standard input and standard output are both available for their usual usage. There is no way of saying the following in the shell without <>:

```
cat food | more - >/dev/tty03 2<>/dev/tty03
```

Another example of <> is one that opens **/dev/tty** on file descriptor 3 for reading and writing:

```
exec 3<> /dev/tty
```

An example of creating a lock file for a critical code region:

```
set -C
until  2> /dev/null > lockfile
do     sleep 30
done
set +C
perform critical function
rm lockfile
```

Since **/dev/null** is not a regular file, no error is generated by redirecting to it in *noclobber* mode.

2.8 Exit Status and Errors

2.8.1 Consequences of Shell Errors

For a non-interactive shell, an error condition encountered by a special built-in (see Section 2.14 on page 67) or other type of utility will cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

Error	Special Built-in	Other Utilities
Shell language syntax error	will exit	will exit
Utility syntax error (option or operand error)	will exit	will not exit
Redirection error	will exit	will not exit
Variable assignment error	will exit	will not exit
Expansion error	will exit	will exit
Command not found	n/a	may exit
Dot script not found	will exit	n/a

An expansion error is one that occurs when the shell expansions defined in Section 2.6 on page 31 are carried out (for example, ${x!y}, because ! is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenisation, rather than during expansion.

If any of the errors shown as ''will (may) exit'' occur in a subshell, the subshell will (may) exit with a non-zero status, but the script containing the subshell will not exit because of the error.

In all of the cases shown in the table, an interactive shell will write a diagnostic message to standard error without exiting.

2.8.2 Exit Status for Commands

Each command has an exit status that can influence the behaviour of other shell commands. The exit status of commands that are not utilities is documented in this section. The exit status of the standard utilities is documented in their respective sections.

If a command is not found, the exit status will be 127. If the command name is found, but it is not an executable utility, the exit status will be 126. Applications that invoke utilities without using the shell should use these exit status values to report similar errors.

If a command fails during word expansion or redirection, its exit status will be greater than zero.

Internally, for purposes of deciding if a command exits with a non-zero exit status, the shell will recognise the entire status value retrieved for the command by the equivalent of the **XSH** specification *wait*( ) function WEXITSTATUS macro. When reporting the exit status with the special parameter ?, the shell will report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal will be reported as greater than 128.

2.9 Shell Commands

This section describes the basic structure of shell commands. The following command descriptions each describe a format of the command that is only used to aid the reader in recognising the command type, and does not formally represent the syntax. Each description discusses the semantics of the command; for a formal definition of the command language, consult Section 2.10 on page 56.

A *command* is one of the following:

- *simple command* (see Section 2.9.1)
- *pipeline* (see Section 2.9.2 on page 49)
- *list* or *compound-list* (see Section 2.9.3 on page 49)
- *compound command* (see Section 2.9.4 on page 52)
- *function definition* (see Section 2.9.5 on page 54).

Unless otherwise stated, the exit status of a command is that of the last simple command executed by the command. There is no limit on the size of any shell command other than that imposed by the underlying system (memory constraints, {ARG_MAX}, and so forth).

2.9.1 Simple Commands

A *simple command* is a sequence of optional variable assignments and redirections, in any sequence, optionally followed by words and redirections, terminated by a control operator.

When a given simple command is required to be executed (that is, when any conditional construct such as an AND-OR list or a **case** statement has not bypassed the simple command), the following expansions, assignments and redirections will all be performed from the beginning of the command text to the end.

1. The words that are recognised as variable assignments or redirections according to Section 2.10.2 on page 56 are saved for processing in steps 3 and 4.
2. The words that are not variable assignments or redirections will be expanded. If any fields remain following their expansion, the first field will be considered the command name and remaining fields will be the arguments for the command.
3. Redirections will be performed as described in Section 2.7 on page 40.
4. Each variable assignment will be expanded for tilde expansion, parameter expansion, command substitution, arithmetic expansion and quote removal prior to assigning the value.

In the preceding list, the order of steps 3 and 4 may be reversed for the processing of special built-in utilities. See Section 2.14 on page 67.

If no command name results, variable assignments will affect the current execution environment. Otherwise, the variable assignments will be exported for the execution environment of the command and will not affect the current execution environment (except for special built-ins). If any of the variable assignments attempt to assign a value to a read-only variable, a variable assignment error will occur. See Section 2.8.1 on page 44 for the consequences of these errors.

If there is no command name, any redirections will be performed in a subshell environment; it is unspecified whether this subshell environment is the same one as that used for a command substitution within the command. (To affect the current execution environment, see the *exec* special built-in.) If any of the redirections performed in the current shell execution environment

fail, the command will immediately fail with an exit status greater than zero, and the shell will write an error message indicating the failure. See Section 2.8.1 on page 44 for the consequences of these failures on interactive and non-interactive shells.

If there is a command name, execution will continue as described in **Command Search and Execution** on page 47. If there is no command name, but the command contained a command substitution, the command will complete with the exit status of the last command substitution performed. Otherwise, the command will complete with a zero exit status.

The following example illustrates both how a variable assignment without a command name affects the current execution environment, and how an assignment with a command name only affects the execution environment of the command.

```
$ x=red
$ echo $x
red
$ export x
$ sh –c 'echo $x'
red
$ x=blue sh –c 'echo $x'
blue
$ echo $x
red
```

This next example illustrates that redirections without a command name are still performed.

```
$ ls foo
ls: foo: no such file or directory
$ > foo
$ ls foo
foo
```

A command without a command name, but one that includes a command substitution, has an exit status of the last command substitution that the shell performed. For example:

```
if     x=$(command)
then   ...
fi
```

An example of redirections without a command name being performed in a subshell shows that the here-document does not disrupt the standard input of the **while** loop:

```
IFS=:
while  read a b
do     echo $a
       <<-eof
       Hello
       eof
done </etc/passwd
```

Some examples of commands without command names in AND-OR lists:

```
> foo || {
       echo "error: foo cannot be created" >&2
       exit 1
}

# set saved if /vmunix.save exists
test -f /vmunix.save && saved=1
```

Command substitution and redirections without command names both occur in subshells, but they are not necessarily the same ones. For example, in:

```
exec 3> file
var=$(echo foo >&3) 3>&1
```

it is unspecified whether **foo** will be echoed to the file or to standard output.

Command Search and Execution

If a simple command results in a command name and an optional list of arguments, the following actions will be performed.

1. If the command name does not contain any slashes, the first successful step in the following sequence will occur:
   a. If the command name matches the name of a special built-in utility, that special built-in utility will be invoked.
   b. If the command name matches the name of a function known to this shell, the function will be invoked as described in Section 2.9.5 on page 54. If the implementation has provided a standard utility in the form of a function, it will not be recognised at this point. It will be invoked in conjunction with the path search in step 1d.
   c. If the command name matches the name of a utility listed in the following table, that utility will be invoked.

alias	*false*	*jobs*	*true*
bg	*fc*	*kill*	*umask*
cd	*fg*	*newgrp*	*unalias*
command	*getopts*	*read*	*wait*

   d. Otherwise, the command will be searched for using the *PATH* environment variable as described in the **XBD** specification, **Chapter 6**, **Environment Variables**:
      i. If the search is successful:
         (a) If the system has implemented the utility as a regular built-in or as a shell function, it will be invoked at this point in the path search.
         (b) Otherwise, the shell will execute the utility in a separate utility environment (see Section 2.12 on page 63) with actions equivalent to calling the **XSH** specification *execve*( ) function with the *path* argument set to the pathname resulting from the search, *arg0* set to the command name, and the remaining arguments set to the operands, if any.

         If the *execve*( ) function fails due to an error equivalent to the **XSH** specification error [ENOEXEC], the shell will execute a command equivalent to having a shell invoked with the command name as its first

operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message, and return an exit status of 126.

Once a utility has been searched for and found (either as a result of this specific search or as part of an unspecified shell startup activity), an implementation may remember its location and need not search for the utility again unless the *PATH* variable has been the subject of an assignment. If the remembered location fails for a subsequent invocation, the shell will repeat the search to find the new location for the utility, if any.

ii. If the search is unsuccessful, the command will fail with an exit status of 127 and the shell will write an error message.

2. If the command name contains at least one slash, the shell will execute the utility in a separate utility environment with actions equivalent to calling the **XSH** specification *execve*( ) function with the *path* and *arg0* arguments set to the command name, and the remaining arguments set to the operands, if any.

   If the *execve*( ) function fails due to an error equivalent to the **XSH** specification error [ENOEXEC], the shell will execute a command equivalent to having a shell invoked with the command name as its first operand, along with any remaining arguments passed along. If the executable file is not a text file, the shell may bypass this command execution, write an error message and return an exit status of 126.

This description requires that the shell can execute shell scripts directly, even if the underlying system does not support the common #! interpreter convention. That is, if file **foo** contains shell commands and is executable, the following will execute **foo**:

```
./foo
```

The sequence selected for the ISO / IEC 9945-2: 1993 standard acknowledges that special built-ins cannot be overridden, but gives the programmer full control over which versions of other utilities are executed. It provides a means of suppressing function lookup (via the *command* utility) for the user's own functions and ensures that any regular built-ins or functions provided by the implementation are under the control of the path search. The mechanisms for associating built-ins or functions with executable files in the path are not specified by this document, but the wording requires that if either is implemented, the application will not be able to distinguish a function or built-in from an executable (other than in terms of performance, presumably). The implementation will ensure that all effects specified by this document resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, and so forth) are identical to those resulting from the invocation of an executable file.

Example: Consider three versions of the *ls* utility:

- The application includes a shell function named *ls*.
- The user writes a utility named *ls* and puts it in **/fred/bin**.
- The example implementation provides *ls* as a regular shell built-in that will be invoked (either by the shell or directly by *exec*) when the path search reaches the directory **/posix/bin**.

If PATH=/posix/bin, various invocations yield different versions of *ls*:

Invocation	Version of *ls*
`ls` (from within application script)	(1) function
`command ls` (from within application script)	(3) built-in
`ls` (from within makefile called by application)	(3) built-in
`system("ls")`	(3) built-in
`PATH="/hsa/bin:$PATH" ls`	(2) user's version

2.9.2 Pipelines

A *pipeline* is a sequence of one or more commands separated by the control operator |. The standard output of all but the last command will be connected to the standard input of the next command.

The format for a pipeline is:

[!**]** *command1* **[** | *command2* …**]**

The standard output of *command1* will be connected to the standard input of *command2*. The standard input, standard output or both of a command will be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see Section 2.7 on page 40).

If the pipeline is not in the background (see Section 2.9.3 on page 50), the shell will wait for the last command specified in the pipeline to complete, and may also wait for all commands to complete.

Exit Status

If the reserved word ! does not precede the pipeline, the exit status will be the exit status of the last command specified in the pipeline. Otherwise, the exit status is the logical NOT of the exit status of the last command. That is, if the last command returns zero, the exit status will be 1; if the last command returns greater than zero, the exit status will be zero.

Because pipeline assignment of standard input or standard output or both takes place before redirection, it can be modified by redirection. For example:

$ *command1* 2>&1 | *command2*

sends both the standard output and standard error of *command1* to the standard input of *command2*.

The reserved word **!** allows more flexible testing using AND and OR lists.

2.9.3 Lists

An *AND-OR-list* is a sequence of one or more pipelines separated by the operators:

&& ||

A *list* is a sequence of one or more AND-OR-lists separated by the operators:

; &

and optionally terminated by:

; & <newline>

The operators && and | | have equal precedence and will be evaluated from beginning to end. For example, both of the following commands write solely **bar** to standard output:

```
false && echo foo || echo bar
true || echo foo && echo bar
```

A ; or newline character terminator will cause the preceding AND-OR-list to be executed sequentially; an & will cause asynchronous execution of the preceding AND-OR-list.

The term *compound-list* is derived from the grammar in Section 2.10 on page 56; it is equivalent to a sequence of *lists*, separated by newline characters, that can be preceded or followed by an arbitrary number of newline characters.

The following is an example that illustrates newline characters in compound-lists:

```
while
       # a couple of newlines

       # a list
       date && who || ls; cat file
       # a couple of newlines

       # another list
       wc file > output & true

do
       # 2 lists
       ls
       cat file
done
```

Asynchronous Lists

If a command is terminated by the control operator ampersand (&), the shell will execute the command asynchronously in a subshell. This means that the shell does not wait for the command to finish before executing the next command.

The format for running a command in the background is:

command1 & [*command2* & ...]

The standard input for an asynchronous list, before any explicit redirections are performed, will be considered to be assigned to a file that has the same properties as **/dev/null**. If it is an interactive shell, this need not happen. In all cases, explicit redirection of standard input will override this activity.

Since the connection of the input to the equivalent of **/dev/null** is considered to occur before redirections, the following script would produce no output:

```
exec < /etc/passwd
cat <&0 &
wait
```

When an element of an asynchronous list (the portion of the list ended by an ampersand, such as *command1*, above) is started by the shell, the process ID of the last command in the asynchronous list element will become known in the current shell execution environment; see Section 2.12 on page 63. This process ID will remain known until:

1. The command terminates and the application waits for the process ID.
2. Another asynchronous list invoked before $! (corresponding to the previous asynchronous list) is expanded in the current execution environment.

The implementation need not retain more than the {CHILD_MAX} most recent entries in its list of known process IDs in the current shell execution environment.

Exit Status: The exit status of an asynchronous list is zero.

Sequential Lists

Commands that are separated by a semicolon (;) will be executed sequentially.

The format for executing commands sequentially is:

command1 **[**; *command2***]** ...

Each command will be expanded and executed in the order specified.

Exit Status: The exit status of a sequential list will be the exit status of the last command in the list.

AND Lists

The control operator && denotes an AND list. The format is:

command1 **[** && *command2***]** ...

First *command1* will be executed. If its exit status is zero, *command2* will be executed, and so on until a command has a non-zero exit status or there are no more commands left to execute. The commands will be expanded only if they are executed.

Exit Status: The exit status of an AND list will be the exit status of the last command that is executed in the list.

OR Lists

The control operator | | denotes an OR List. The format is:

command1 **[** || *command2***]** ...

First, *command1* will be executed. If its exit status is non-zero, *command2* will be executed, and so on until a command has a zero exit status or there are no more commands left to execute.

Exit Status: The exit status of an OR list will be the exit status of the last command that is executed in the list.

2.9.4 Compound Commands

The shell has several programming constructs that are *compound commands*, which provide control flow for commands. Each of these compound commands has a reserved word or control operator at the beginning, and a corresponding terminator reserved word or operator at the end. In addition, each can be followed by redirections on the same line as the terminator. Each redirection will apply to all the commands within the compound command that do not explicitly override that redirection.

Grouping Commands

The format for grouping commands is as follows:

(*compound-list*) Execute *compound-list* in a subshell environment; see Section 2.12 on page 63. Variable assignments and built-in commands that affect the environment will not remain in effect after the list finishes.

{ *compound-list*;} Execute *compound-list* in the current process environment. The semicolon shown here is an example of a control operator delimiting the } reserved word. Other delimiters are possible, as shown in Section 2.10 on page 56; a newline character is frequently used.

Exit Status: The exit status of a grouping command will be the exit status of *list*.

For Loop

The **for** loop will execute a sequence of commands for each member in a list of *items*. The **for** loop requires that the reserved words **do** and **done** be used to delimit the sequence of commands.

The format for the **for** loop is as follows:

```
for name [ in word ... ]
do
        compound-list
done
```

First, the list of words following **in** will be expanded to generate a list of items. Then, the variable *name* will be set to each item, in turn, and the *compound-list* executed each time. If no items result from the expansion, the *compound-list* will not be executed. Omitting:

```
in word…
```

is equivalent to:

```
in "$@"
```

The format is shown with generous usage of newline characters. See the grammar in Section 2.10 on page 56 for a precise description of where newline characters and semicolons can be interchanged.

Exit Status: The exit status of a **for** command will be the exit status of the last command that executes. If there are no items, the exit status will be zero.

Case Conditional Construct

The conditional construct **case** will execute the *compound-list* corresponding to the first one of several *patterns* (see Section 2.13 on page 64) that is matched by the string resulting from the tilde expansion, parameter expansion, command substitution, and arithmetic expansion and quote removal of the given word. The reserved word **in** will denote the beginning of the patterns to be matched. Multiple patterns with the same *compound-list* are delimited by the | symbol. The control operator) terminates a list of patterns corresponding to a given action. The *compound-list* for each list of patterns is terminated with ;;. The **case** construct terminates with the reserved word **esac** (**case** reversed).

The format for the **case** construct is as follows:

```
case word in
     [(]pattern1)               compound-list;;
     [(]pattern2|pattern3)      compound-list;;
     ...
esac
```

The ;; is optional for the last *compound-list*.

In order from the beginning to the end of the **case** statement, each *pattern* that labels a *compound-list* is subjected to tilde expansion, parameter expansion, command substitution and arithmetic expansion, and the result of these expansions is compared against the expansion of *word*, according to the rules described in Section 2.13 on page 64 (which also describes the effect of quoting parts of the pattern). After the first match, no more patterns are expanded, and the *compound-list* is executed. The order of expansion and comparison of multiple *pattern*s that label a *compound-list* statement is unspecified.

Exit Status: The exit status of **case** is zero if no patterns are matched. Otherwise, the exit status will be the exit status of the last command executed in the *compound-list*.

The pattern *, given as the last pattern in a **case** construct, is equivalent to the default case in a C-language **switch** statement.

The grammar shows that reserved words can be used as patterns, even if one is the first word on a line. Obviously, the reserved word **esac** cannot be used in this manner.

If Conditional Construct

The **if** command will execute a *compound-list* and use its exit status to determine whether to execute another *compound-list*.

The format for the **if** construct is as follows:

```
if compound-list
then
       compound-list
[elif compound-list
then
       compound-list] ...
[else
       compound-list]
fi
```

The **if** *compound-list* is executed; if its exit status is zero, the **then** *compound-list* is executed and the command will complete. Otherwise, each **elif** *compound-list* is executed, in turn, and if its exit status is zero, the *then compound-list* is executed and the command will complete.

Otherwise, the **else** *compound-list* is executed.

Exit Status: The exit status of the **if** command will be the exit status of the *then* or **else** *compound-list* that was executed, or zero, if none was executed.

While Loop

The **while** loop continuously will execute one *compound-list* as long as another *compound-list* has a zero exit status.

The format of the **while** loop is as follows:

```
while compound-list-1
do
        compound-list-2
done
```

The *compound-list-1* will be executed, and if it has a non-zero exit status, the **while** command will complete. Otherwise, the *compound-list-2* will be executed, and the process will repeat.

Exit Status: The exit status of the **while** loop will be the exit status of the last *compound-list-2* executed, or zero if none was executed.

Until Loop

The **until** loop continuously will execute one *compound-list* as long as another *compound-list* has a non-zero exit status.

The format of the **until** loop is as follows:

```
until compound-list-1
do
        compound-list-2
done
```

The *compound-list-1* will be executed, and if it has a zero exit status, the **until** command will complete. Otherwise, the *compound-list-2* will be executed, and the process will repeat.

Exit Status: The exit status of the **until** loop will be the exit status of the last *compound-list-2* executed, or zero if none was executed.

2.9.5 Function Definition Command

A function is a user-defined name that is used as a simple command to call a compound command with new positional parameters. A function is defined with a *function definition command*.

The format of a function definition command is as follows:

```
fname() compound-command[io-redirect ...]
```

The function is named *fname*; it must be a name (see **name** in the **XBD** specification, **Chapter 2**, **Glossary**). An implementation may allow other characters in a function name as an extension. The implementation will maintain separate name spaces for functions and variables.

The () in the function definition command consists of two operators. Therefore, intermixing blank characters with the *fname*, (and) is allowed, but unnecessary.

The argument *compound-command* represents a compound command, as described in Section 2.9.4 on page 52.

When the function is declared, none of the expansions in Section 2.6 on page 31 will be performed on the text in *compound-command* or *io-redirect*; all expansions will be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* will be performed during the execution of the function itself, not the function definition. See Section 2.8.1 on page 44 for the consequences of failures of these operations on interactive and non-interactive shells.

When a function is executed, it will have the syntax-error and variable-assignment properties described for special built-in utilities in the enumerated list at the beginning of Section 2.14 on page 67.

The *compound-command* will be executed whenever the function name is specified as the name of a simple command (see **Command Search and Execution** on page 47). The operands to the command temporarily will become the positional parameters during the execution of the *compound-command*; the special parameter # will also be changed to reflect the number of operands. The special parameter 0 will be unchanged. When the function completes, the values of the positional parameters and the special parameter # will be restored to the values they had before the function was executed. If the special built-in *return* is executed in the *compound-command*, the function will complete and execution will resume with the next command after the function call.

An example of how a function definition can be used wherever a simple command is allowed:

```
# If variable i is equal to "yes",
# define function foo to be ls -l
#
[ "$i" = yes ] && foo() {
       ls -l
}
```

Exit Status

The exit status of a function definition will be zero if the function was declared successfully; otherwise, it will be greater than zero. The exit status of a function invocation will be the exit status of the last command executed by the function.

2.10 Shell Grammar

The following grammar defines the Shell Command Language. This formal syntax takes precedence over the preceding text syntax description.

2.10.1 Shell Grammar Lexical Conventions

The input language to the shell must be first recognised at the character level. The resulting tokens will be classified by their immediate context according to the following rules (applied in order). These rules are used to determine what a ‘‘token’’ that is subject to parsing at the token level is. The rules for token recognition in Section 2.3 on page 23 will apply.

1. A newline character will be returned as the token identifier **NEWLINE**.
2. If the token is an operator, the token identifier for that operator will result.
3. If the string consists solely of digits and the delimiter character is one of < or >, the token identifier **IO_NUMBER** will be returned.
4. Otherwise, the token identifier **TOKEN** will result.

Further distinction on **TOKEN** is context-dependent. It may be that the same **TOKEN** yields **WORD**, a **NAME**, an **ASSIGNMENT**, or one of the reserved words below, dependent upon the context. Some of the productions in the grammar below are annotated with a rule number from the following list. When a **TOKEN** is seen where one of those annotated productions could be used to reduce the symbol, the applicable rule will be applied to convert the token identifier type of the **TOKEN** to a token identifier acceptable at that point in the grammar. The reduction will then proceed based upon the token identifier type yielded by the rule applied. When more than one rule applies, the highest numbered rule will apply (which in turn may refer to another rule). (Note that except in rule 7, the presence of an = in the token has no effect.)

The **WORD** tokens will have the word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

2.10.2 Shell Grammar Rules

1. [Command Name]
   When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word will result. Otherwise, the token **WORD** will be returned. Also, if the parser is in any state where only a reserved word could be the next correct token, proceed as above. This rule applies rather narrowly: when a compound list is terminated by some clear delimiter (such as the closing **fi** of an inner **if_clause**) then it would apply; where the compound list might continue (as in after a ;), rule 7a (and consequently the first sentence of this rule) would apply. In many instances the two conditions are identical, but this part of this rule does not give licence to treating a **WORD** as a reserved word unless it is in a place where a reserved word must appear.

   Note: Because at this point quote marks are retained in the token, quoted strings cannot be recognised as reserved words. This rule also implies that reserved words will not be recognised except in certain positions in the input, such as after a newline character or semicolon; the grammar presumes that if the reserved word is intended, it will be properly delimited by the user, and does not attempt to reflect that requirement directly. Also note that line joining is done before tokenisation, as described in Section 2.2.1 on page 20, so escaped newlines are already removed at this point.

Rule 1 is not directly referenced in the grammar, but is referred to by other rules, or applies globally.

2. [Redirection to or from filename]
   The expansions specified in Section 2.7 on page 40 will occur. As specified there, exactly one field can result (or the result is unspecified), and there are additional requirements on pathname expansion.

3. [Redirection from here-document]
   Quote removal will be applied to the word to determine the delimiter that will be used to find the end of the here-document that begins after the next newline character.

4. [Case statement termination]
   When the **TOKEN** is exactly the reserved word **Esac**, the token identifier for **Esac** will result. Otherwise, the token **WORD** will be returned.

5. [NAME in **for**]
   When the **TOKEN** meets the requirements for a name (see **name** in the **XBD** specification, **Chapter 2**, **Glossary**), the token identifier **NAME** will result. Otherwise, the token **WORD** will be returned.

6. [Third word of **for** and **case**]
   When the **TOKEN** is exactly the reserved word **In**, the token identifier for **In** will result. Otherwise, the token **WORD** will be returned. (As indicated in the grammar, a **linebreak** precedes the token **In**. If newline characters are present at the indicated location, it is the token after them that is treated in this fashion.)

7. [Assignment preceding command name]

   a. [When the first word]
      If the **TOKEN** does not contain the character =, rule 1 will be applied. Otherwise, 7b will be applied.

   b. [Not the first word]
      If the **TOKEN** contains the equal sign character:

      — If it begins with =, the token **WORD** will be returned.

      — If all the characters preceding = form a valid name (see **name** in the **XSH** specification), the token **ASSIGNMENT_WORD** will be returned. (Quoted characters cannot participate in forming a valid name.)

      — Otherwise, it is unspecified whether it is **ASSIGNMENT_WORD** or **WORD** that is returned.

   Assignment to the **NAME** will occur as specified in Section 2.9.1 on page 45.

8. [NAME in function]
   When the **TOKEN** is exactly a reserved word, the token identifier for that reserved word will result. Otherwise, when the **TOKEN** meets the requirements for a name (see **Name**), the token identifier **NAME** will result. Otherwise, rule 7 will apply.

9. [Body of function]
   Word expansion and assignment will never occur, even when required by the rules above, when this rule is being parsed. Each **TOKEN** that might either be expanded or have assignment applied to it will instead be returned as a single **WORD** consisting only of characters that are exactly the token described in Section 2.3 on page 23.

```
/*      ------------------------------------------------------
        The grammar symbols
        ------------------------------------------------------ */

%token  WORD
%token  ASSIGNMENT_WORD
%token  NAME
%token  NEWLINE
%token  IO_NUMBER

/*      The following are the operators mentioned above.        */

%token  AND_IF OR_IF DSEMI
/*      '&&'   '||'  ';;'                                       */

%token  DLESS DGREAT LESSAND GREATAND LESSGREAT DLESSDASH
/*      '<<'  '>>'   '<&'    '>&'     '<>'     '<<-'           */

%token  CLOBBER
/*      '>|'                                                    */

/*      The following are the reserved words                    */

%token   If   Then   Else   Elif   Fi   Do   Done
/*      'if' 'then' 'else' 'elif' 'fi' 'do' 'done'              */

%token   Case   Esac   While   Until   For
/*      'case' 'esac' 'while' 'until' 'for'                     */

/*      These are reserved words, not operator tokens, and are
        recognised when reserved words are recognised.          */

%token  Lbrace Rbrace Bang
/*       '{'    '}'    '!'                                     */

%token  In
/*      'in'                                                    */

/*      -------------------------------------------------------
        The Grammar
        ------------------------------------------------------- */

%start  complete_command

%%

complete_command    : list separator
                    | list
                    ;
list                : list separator_op and_or
                    |                   and_or
                    ;
and_or              :                         pipeline
                    | and_or AND_IF linebreak pipeline
                    | and_or OR_IF  linebreak pipeline
                    ;
```

```
pipeline            :      pipe_sequence
                    | Bang pipe_sequence
                    ;
pipe_sequence       :                             command
                    | pipe_sequence '|' linebreak command
                    ;
command             : simple_command
                    | compound_command
                    | compound_command redirect_list
                    | function_definition
                    ;
compound_command    : brace_group
                    | subshell
                    | for_clause
                    | case_clause
                    | if_clause
                    | while_clause
                    | until_clause
                    ;
subshell            : '(' compound_list ')'
                    ;
compound_list       :              term
                    | newline_list term
                    |              term separator
                    | newline_list term separator
                    ;
term                : term separator and_or
                    |                and_or
                    ;
for_clause          : For name linebreak                            do_group
                    | For name linebreak in wordlist sequential_sep do_group
                    ;
name                : NAME                     /* Apply rule 5 */
                    ;
in                  : In                       /* Apply rule 6 */
                    ;
wordlist            : wordlist WORD
                    |          WORD
                    ;
case_clause         : Case WORD linebreak in linebreak case_list Esac
                    | Case WORD linebreak in linebreak           Esac
                    ;
case_list           : case_list case_item
                    |           case_item
                    ;
case_item           :     pattern ')' linebreak     DSEMI linebreak
                    |     pattern ')' compound_list DSEMI linebreak
                    | '(' pattern ')' linebreak     DSEMI linebreak
                    | '(' pattern ')' compound_list DSEMI linebreak
                    ;
pattern             :             WORD           /* Apply rule 4 */
```

```
                    | pattern '|' WORD         /* Do not apply rule (4) */
                    ;
if_clause           : If compound_list Then compound_list else_part Fi
                    | If compound_list Then compound_list           Fi
                    ;
else_part           : Elif compound_list Then else_part
                    | Else compound_list
                    ;
while_clause        : While compound_list do_group
                    ;
until_clause        : Until compound_list do_group
                    ;
function_definition : fname '(' ')' linebreak function_body
                    ;
function_body       : compound_command                /* Apply rule 9 */
                    | compound_command redirect_list  /* Apply rule 9 */
                    ;
fname               : NAME                            /* Apply rule 8 */
                    ;
brace_group         : Lbrace compound_list Rbrace
                    ;
do_group            : Do compound_list Done
                    ;
simple_command      : cmd_prefix cmd_word cmd_suffix
                    | cmd_prefix cmd_word
                    | cmd_prefix
                    | cmd_name cmd_suffix
                    | cmd_name
                    ;
cmd_name            : WORD                   /* Apply rule 7a */
                    ;
cmd_word            : WORD                   /* Apply rule 7b */
                    ;
cmd_prefix          :            io_redirect
                    | cmd_prefix io_redirect
                    |            ASSIGNMENT_WORD
                    | cmd_prefix ASSIGNMENT_WORD
                    ;
cmd_suffix          :            io_redirect
                    | cmd_suffix io_redirect
                    |            WORD
                    | cmd_suffix WORD
                    ;
redirect_list       :               io_redirect
                    | redirect_list io_redirect
                    ;
io_redirect         :           io_file
                    | IO_NUMBER io_file
                    |           io_here
                    | IO_NUMBER io_here
```

```
                    ;
io_file             : '<'       filename
                    | LESSAND   filename
                    | '>'       filename
                    | GREATAND  filename
                    | DGREAT    filename
                    | LESSGREAT filename
                    | CLOBBER   filename
                    ;
filename            : WORD                      /* Apply rule 2 */
                    ;
io_here             : DLESS     here_end
                    | DLESSDASH here_end
                    ;
here_end            : WORD                      /* Apply rule 3 */
                    ;
newline_list        :              NEWLINE
                    | newline_list NEWLINE
                    ;
linebreak           : newline_list
                    | /* empty */
                    ;
separator_op        : '&'
                    | ';'
                    ;
separator           : separator_op linebreak
                    | newline_list
                    ;
sequential_sep      : ';' linebreak
                    | newline_list
                    ;
```

There are several subtle aspects of this grammar where conventional usage implies rules about the grammar that in fact are not true.

For **compound_list**, only the forms that end in a **separator** allow a reserved word to be recognised, so usually only a **separator** can be used where a compound list precedes a reserved word (such as **Then**, **Else**, **Do** and **Rbrace**). Explicitly requiring a separator would disallow such valid (if rare) statements as:

```
if (false) then (echo x) else (echo y) fi
```

See the Note under special grammar rule 1.

Note that the bodies of here-documents are handled by token recognition (see Section 2.3 on page 23) and do not appear in the grammar directly. (However, the here-document I/O redirection operator is handled as part of the grammar.)

The start symbol of the grammar (**complete_command**) represents either input from the command line or a shell script. It is repeatedly applied by the interpreter to its input, and represents a single chunk of that input as seen by the interpreter.

2.11 Signals and Error Handling

When a command is in an asynchronous list, the shell will prevent SIGQUIT and SIGINT signals from the keyboard from interrupting the command. Otherwise, signals will have the values inherited by the shell from its parent (see also the *trap* special built-in).

When a signal for which a trap has been set is received while the shell is waiting for the completion of a utility executing a foreground command, the trap associated with that signal will not be executed until after the foreground command has completed. When the shell is waiting, by means of the *wait* utility, for asynchronous commands to complete, the reception of a signal for which a trap has been set will cause the *wait* utility to return immediately with an exit status >128, immediately after which the trap associated with that signal will be taken.

If multiple signals are pending for the shell for which there are associated trap actions, the order of execution of trap actions is unspecified.

2.12 Shell Execution Environment

A shell execution environment consists of the following:

- open files inherited upon invocation of the shell, plus open files controlled by *exec*
- working directory as set by *cd*
- file creation mask set by *umask*
- current traps set by *trap*
- shell parameters that are set by variable assignment (see the *set* special built-in) or from the **XSH** specification environment inherited by the shell when it begins (see the *export* special built-in)
- shell functions (see Section 2.9.5 on page 54)
- options turned on at invocation or by *set*
- process IDs of the last commands in asynchronous lists known to this shell environment; see Section 2.9.3 on page 50
- shell aliases (see Section 2.3.1 on page 24).

Utilities other than the special built-ins (see Section 2.14 on page 67) will be invoked in a separate environment that consists of the following. The initial value of these objects will be the same as that for the parent shell, except as noted below.

- open files inherited on invocation of the shell, open files controlled by the *exec* special built-in plus any modifications and additions specified by any redirections to the utility
- current working directory
- file creation mask
- if the utility is a shell script, traps caught by the shell will be set to the default values and traps ignored by the shell will be set to be ignored by the utility; if the utility is not a shell script, the trap actions (default or ignore) will be mapped into the appropriate signal handling actions for the utility
- variables with the *export* attribute, along with those explicitly exported for the duration of the command, will be passed to the utility as **XSH** specification environment variables.

The environment of the shell process will not be changed by the utility unless explicitly specified by the utility description (for example, *cd* and *umask*).

A subshell environment will be created as a duplicate of the shell environment, except that signal traps set by that shell environment will be set to the default values. Changes made to the subshell environment will not affect the shell environment. Command substitution, commands that are grouped with parentheses and asynchronous lists will be executed in a subshell environment. Additionally, each command of a multi-command pipeline is in a subshell environment; as an extension, however, any or all commands in a pipeline may be executed in the current environment. All other commands will be executed in the current shell environment.

Some systems have implemented the last stage of a pipeline in the current environment so that commands such as:

```
command | read foo
```

set variable **foo** in the current environment. This extension is allowed, but not required; therefore, a shell programmer should consider a pipeline to be in a subshell environment, but not depend on it.

2.13 Pattern Matching Notation

The pattern matching notation described in this section is used to specify patterns for matching strings in the shell. Historically, pattern matching notation is related to, but slightly different from, the regular expression notation described in the **XBD** specification, **Chapter 7**, **Regular Expressions**. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation.

2.13.1 Patterns Matching a Single Character

The following *patterns matching a single character* match a single character: *ordinary characters*, *special pattern characters* and *pattern bracket expressions*. The pattern bracket expression will also match a single collating element.

An ordinary character is a pattern that matches itself. It can be any character in the supported character set except for NUL, those special shell characters in Section 2.2 on page 20 that require quoting, and the following three special pattern characters. Matching is based on the bit pattern used for encoding the character, not on the graphic representation of the character. If any character (ordinary, shell special or pattern special) is quoted, that pattern will match the character itself. The shell special characters always require quoting.

When unquoted and outside a bracket expression, the following three characters will have special meaning in the specification of patterns:

? A question-mark is a pattern that will match any character.

* An asterisk is a pattern that will match multiple characters, as described in Section 2.13.2 on page 65.

[The open bracket will introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in the **XBD** specification, **Section 7.3.5**, **RE Bracket Expression** also applies to the pattern bracket expression, except that the exclamation-mark character (!) replaces the circumflex character (^) in its role in a *non-matching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

The restriction on a circumflex in a bracket expression is to allow implementations that support pattern matching using the circumflex as the negation character in addition to the exclamation-mark. A portable application must use something like [\^!] to match either character.

When pattern matching is used where shell quote removal is not performed (such as in the argument to the *find* –**name** primary when *find* is being called using one of the **XSH** specification *exec* functions, or in the *pattern* argument to the *fnmatch*( ) function), special characters can be escaped to remove their special meaning by preceding them with a backslash character. This escaping backslash will be discarded. The sequence \\ represents one literal backslash. All of the requirements and effects of quoting on ordinary, shell special and special pattern characters will apply to escaping in this context.

Both quoting and escaping are described here because pattern matching must work in three separate circumstances:

- Calling directly upon the shell, such as in pathname expansion or in a **case** statement. All of the following will match the string or file **abc:**

```
abc  "abc"  a"b"c  a\bc  a[b]c  a["b"]c  a[\b]c  a["\b"]c  a?c  a*c
```

The following will not:

```
"a?c"  a\*c  a\[b]c
```

- Calling a utility or function without going through a shell, as described for *find* and the **XSH** specification function *fnmatch*( ).
- Calling utilities such as *find*, *cpio*, *tar* or *pax* through the shell command line. In this case, shell quote removal is performed before the utility sees the argument. For example, in:

  ```
  find /bin -name "e\c[\h]o" -print
  ```

  after quote removal, the backslashes are presented to *find* and it treats them as escape characters. Both precede ordinary characters, so the **c** and **h** represent themselves and **echo** would be found on many historical systems (that have it in **/bin**). To find a filename that contained shell special characters or pattern characters, both quoting and escaping are required, such as:

  ```
  pax -r ... "*a\(\?"
  ```

  to extract a filename ending with **a(?**.

Conforming applications are required to quote or escape the shell special characters (sometimes called metacharacters). If used without this protection, syntax errors can result or implementation extensions can be triggered. For example, the KornShell supports a series of extensions based on parentheses in patterns.

2.13.2 Patterns Matching Multiple Characters

The following rules are used to construct *patterns matching multiple characters* from *patterns matching a single character*:

1. The asterisk (*) is a pattern that will match any string, including the null string.
2. The concatenation of *patterns matching a single character* is a valid pattern that will match the concatenation of the single characters or collating elements matched by each of the concatenated patterns.
3. The concatenation of one or more *patterns matching a single character* with one or more asterisks is a valid pattern. In such patterns, each asterisk will match a string of zero or more characters, matching the greatest possible number of characters that still allows the remainder of the pattern to match the string.

Since each asterisk matches zero or more occurrences, the patterns a*b and a**b have identical functionality.

Examples:

a[bc]	matches the strings **ab** and **ac**.
a*d	matches the strings **ad**, **abd** and **abcd**, but not the string **abc**.
a*d*	matches the strings **ad**, **abcd**, **abcdef**, **aaaad** and **adddd**.
*a*d	matches the strings **ad**, **abcd**, **efabcd**, **aaaad** and **adddd**.

2.13.3 Patterns Used for Filename Expansion

The rules described so far in Section 2.13.1 on page 64 and Section 2.13.2 on page 65 are qualified by the following rules that apply when pattern matching notation is used for filename expansion.

1. The slash character in a pathname must be explicitly matched by using one or more slashes in the pattern; it cannot be matched by the asterisk or question-mark special characters or by a bracket expression. Slashes in the pattern are identified before bracket expressions; thus, a slash cannot be included in a pattern bracket expression used for filename expansion. For example, the pattern a[b/c]d will not match such pathnames as **abd** or **a/d**. It will only match a pathname of literally **a[b/c]d**.

2. If a filename begins with a period (.) the period must be explicitly matched by using a period as the first character of the pattern or immediately following a slash character. The leading period will not be matched by:

   - the asterisk or question-mark special characters
   - a bracket expression containing a non-matching list, such as:

     ```
     [!a]
     ```

     a range expression, such as:

     ```
     [%–0]
     ```

     or a character class expression, such as:

     ```
     [[:punct:]]
     ```

   It is unspecified whether an explicit period in a bracket expression matching list, such as:

   ```
   [.abc]
   ```

   can match a leading period in a filename.

3. Specified patterns are matched against existing filenames and pathnames, as appropriate. Each component that contains a pattern character requires read permission in the directory containing that component. Any component, except the last, that does not contain a pattern character requires search permission. For example, given the pattern:

   ```
   /foo/bar/x*/bam
   ```

   search permission is needed for directories **/** and **foo**, search and read permissions are needed for directory **bar**, and search permission is needed for each **x*** directory. If the pattern matches any existing filenames or pathnames, the pattern will be replaced with those filenames and pathnames, sorted according to the collating sequence in effect in the current locale. If the pattern contains an invalid bracket expression or does not match any existing filenames or pathnames, the pattern string is left unchanged.

2.14 Special Built-in Utilities

The following *special built-in* utilities will be supported in the shell command language. The output of each command, if any, will be written to standard output, subject to the normal redirection and piping possible with all commands.

The term *built-in* implies that the shell can execute the utility directly and does not need to search for it. An implementation can choose to make any utility a built-in; however, the special built-in utilities described here differ from regular built-in utilities in two respects:

- A syntax error in a special built-in utility may cause a shell executing that utility to abort, while a syntax error in a regular built-in utility will not cause a shell executing that utility to abort. (See Section 2.8.1 on page 44 for the consequences of errors on interactive and non-interactive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value will be non-zero.
- Variable assignments specified with special built-in utilities will remain in effect after the built-in completes; this is not the case with a regular built-in or other utility.

The special built-in utilities in this section need not be provided in a manner accessible via the **XSH** specification *exec* family of functions.

Some of the special built-ins are described as conforming to the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. For those that are not, the requirement in the **XBD** specification, **Section 2.1**, **Utility Description Defaults** that `--` be recognised as a first argument to be discarded does not apply and a portable application must not use that argument.

2.14.1 break — Exit From for, while or until Loop

SYNOPSIS

```
break [n]
```

DESCRIPTION

Exit from the smallest enclosing **for**, **while** or **until** loop, if any; or from the *n*th enclosing loop if *n* is specified. The value of *n* is an unsigned decimal integer greater than or equal to 1. The default is equivalent to *n*=1. If *n* is greater than the number of enclosing loops, the last enclosing loop is exited from. Execution will continue with the command immediately following the loop.

EXAMPLES

```
for i in *
do
       if test –d "$i"
       then break
       fi
done
```

EXIT STATUS

0 Successful completion.
>0 The *n* value was not an unsigned decimal integer greater than or equal to 1.

2.14.2 colon — Null Utility

SYNOPSIS

```
: [argument ...]
```

DESCRIPTION

This utility will only expand command *argument*s. It is used when a command is needed, as in the *then* condition of an **if** command, but nothing is to be done by the command.

EXAMPLES

```
: ${X=abc}
if     false
then   :
else   echo $X
fi
abc
```

As with any of the special built-ins, the null utility can also have variable assignments and redirections associated with it, such as:

```
x=y : > z
```

which sets variable **x** to the value y (so that it persists after the null utility completes) and creates or truncates file **z**.

EXIT STATUS

Zero.

2.14.3 continue — Continue for, while or until Loop

SYNOPSIS

```
continue [n]
```

DESCRIPTION

The *continue* utility will return to the top of the smallest enclosing **for**, **while** or **until** loop, or to the top of the *n*th enclosing loop, if *n* is specified. This involves repeating the condition list of a **while** or **until** loop or performing the next assignment of a **for** loop, and reexecuting the loop if appropriate.

The value of *n* is a decimal integer greater than or equal to 1. The default is equivalent to *n*=1. If *n* is greater than the number of enclosing loops, the last enclosing loop is used.

EXAMPLES

```
for i in *
do
       if test –d "$i"
       then continue
       fi
done
```

EXIT STATUS

0 Successful completion.
>0 The *n* value was not an unsigned decimal integer greater than or equal to 1.

2.14.4 dot — Execute Commands in Current Environment

SYNOPSIS

```
. file
```

DESCRIPTION

The shell will execute commands from the *file* in the current environment.

If *file* does not contain a slash, the shell will use the search path specified by *PATH* to find the directory containing *file*. Unlike normal command search, however, the file searched for by the *dot* utility need not be executable. If no readable file is found, a non-interactive shell will abort; an interactive shell will write a diagnostic message to standard error, but this condition will not be considered a syntax error.

EXAMPLES

```
cat foobar
foo=hello bar=world
. foobar
echo $foo $bar
hello world
```

EXIT STATUS

Returns the value of the last command executed, or a zero exit status if no command is executed.

2.14.5 eval — Construct Command by Concatenating Arguments

SYNOPSIS

```
eval [argument ...]
```

DESCRIPTION

The *eval* utility will construct a command by concatenating *argument*s together, separating each with a space character. The constructed command will be read and executed by the shell.

EXAMPLES

```
foo=10 x=foo
y='$'$x
echo $y
$foo
eval y='$'$x
echo $y
10
```

EXIT STATUS

If there are no *argument*s, or only null arguments, *eval* will return a zero exit status; otherwise, it will return the exit status of the command defined by the string of concatenated *argument*s separated by spaces.

2.14.6 exec — Execute Commands and Open, Close or Copy File Descriptors

SYNOPSIS

exec [*command* [*argument* ...]]

DESCRIPTION

The *exec* utility will open, close or copy file descriptors as specified by any redirections as part of the command.

If *exec* is specified without *command* or *arguments*, and any file descriptors with numbers > 2 are opened with associated redirection statements, it is unspecified whether those file descriptors remain open when the shell invokes another utility. Scripts concerned that child shells could misuse open file descriptors can always close them explicitly, as shown in one of the following examples.

If *exec* is specified with *command*, it will replace the shell with *command* without creating a new process. If *argument*s are specified, they are arguments to *command*. Redirection will affect the current shell execution environment.

EXAMPLES

Open **readfile** as file descriptor 3 for reading:

```
exec 3< readfile
```

Open **writefile** as file descriptor 4 for writing:

```
exec 4> writefile
```

Make unit 5 a copy of unit 0:

```
exec 5<&0
```

Close file unit 3:

```
exec 3<&–
```

Cat the file **maggie** by replacing the current shell with the *cat* utility:

```
exec cat maggie
```

EXIT STATUS

If *command* is specified, *exec* will not return to the shell; rather, the exit status of the process will be the exit status of the program implementing *command*, which overlaid the shell. If *command* is not found, the exit status will be 127. If *command* is found, but it is not an executable utility, the exit status will be 126. If a redirection error occurs (see Section 2.8.1 on page 44), the shell will exit with a value in the range 1–125. Otherwise, *exec* will return a zero exit status.

2.14.7 exit — Cause the Shell to Exit

SYNOPSIS

`exit [`*n*`]`

DESCRIPTION

The *exit* utility causes the shell to exit with the exit status specified by the unsigned decimal integer *n*. If *n* is specified, but its value is not between 0 and 255 inclusively, the exit status is undefined.

As explained in other sections, certain exit status values have been reserved for special uses and should be used by applications only for those purposes:

126 A command to be executed was found, but the file was not an executable utility.

127 A command to be executed was not found.

>128 A command was interrupted by a signal.

A trap on **EXIT** will be executed before the shell terminates, except when the *exit* utility is invoked in that trap itself, in which case the shell will exit immediately.

EXAMPLES

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```

EXIT STATUS

The exit status will be *n*, if specified. Otherwise, the value will be the exit value of the last command executed, or zero if no command was executed. When *exit* is executed in a trap action, the last command is considered to be the command that executed immediately preceding the trap action.

2.14.8 export — Set Export Attribute for Variables

SYNOPSIS

```
export  name[=word]...
export –p
```

DESCRIPTION

The shell will give the export attribute to the variables corresponding to the specified *name*s, which will cause them to be in the environment of subsequently executed commands.

The *export* special built-in supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

When **–p** is specified, *export* will write to the standard output the names and values of all exported variables, in the following format:

```
"export %s=%s\n", <name>, <value>
```

The **–p** option allows portable access to the values that can be saved and then later restored using, for instance, a dot script.

The shell will format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same exporting results.

When no arguments are given, the results are unspecified.

EXAMPLES

Export *PWD* and *HOME* variables:

```
export PWD HOME
```

Set and export the *PATH* variable:

```
export PATH=/local/bin:$PATH
```

Save and restore all exported variables:

```
export -p > temp-file
unset a lot of variables
... processing
. temp-file
```

EXIT STATUS

Zero.

2.14.9 readonly — Set Read-only Attribute for Variables

SYNOPSIS

```
readonly  name[=word]...
readonly –p
```

DESCRIPTION

The variables whose *name*s are specified will be given the *readonly* attribute. The values of variables with the read-only attribute cannot be changed by subsequent assignment, nor can those variables be unset by the *unset* utility.

Some versions of the shell exist that preserve the read-only attribute across separate invocations. This document allows this behaviour, but does not require it.

The *readonly* special built-in supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

When –**p** is specified, *readonly* will write to the standard output the names and values of all read-only variables, in the following format:

```
"readonly %s=%s\n", <name>, <value>
```

The shell will format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same attribute-setting results.

The –**p** option allows portable access to the values that can be saved and then later restored using, for instance, a dot script. (See a related example under *export*.)

EXAMPLES

```
readonly HOME PWD
```

EXIT STATUS

Zero.

2.14.10 return — Return from a Function

SYNOPSIS

```
return  [n]
```

DESCRIPTION

The return utility will cause the shell to stop executing the current function or dot script. If the shell is not currently executing a function or dot script, the results are unspecified.

The results of returning a number greater than 255 are undefined because of differing practices in the various historical implementations. Some shells AND out all but the low-order 8 bits; others allow larger values, but not of unlimited size.

See the discussion of appropriate exit status values under *exit*.

EXIT STATUS

The value of the special parameter ? will be set to *n*, an unsigned decimal integer, or to the exit status of the last command executed if *n* is not specified. If the value of *n* is greater than 255, the results are undefined. When *return* is executed in a trap action, the last command is considered to be the command that executed immediately preceding the trap action.

2.14.11 set — Set or Unset Options and Positional Parameters

SYNOPSIS

EX `set [-abCefmnuvx][-h][-o option][argument...]`

EX `set [+abCefmnuvx][+h][-o option][argument...]`

`set --[argument...]`

OB `set -[argument...]`

DESCRIPTION

If no options or *argument*s are specified, *set* will write the names and values of all shell variables in the collation sequence of the current locale. Each *name* will start on a separate line, using the format:

```
"%s=%s\n", <name>, <value>
```

The *value* string will be written with appropriate quoting so that it is suitable for reinput to the shell, setting or resetting, as far as possible, the variables that are currently set. Read-only variables cannot be reset. See the description of shell quoting in Section 2.2 on page 20.

When options are specified, they will set or unset attributes of the shell, as described below. When *argument*s are specified, they will cause positional parameters to be set or unset, as described below. Setting or unsetting attributes and positional parameters are not necessarily related actions, but they can be combined in a single invocation of *set*.

The *set* special built-in supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** except that options can be specified with either a leading hyphen (meaning enable the option) or plus sign (meaning disable it).

Implementations support the options in the following list in both their hyphen and plus-sign forms. These options can also be specified as options to *sh*.

–**a** When this option is on, the export attribute will be set for each variable to which an assignment is performed. (See **Assignment** in the **XBD** specification, **Chapter 2**, **Glossary**.) If the assignment precedes a utility name in a command, the export attribute will not persist in the current execution environment after the utility completes, with the exception that preceding one of the special built-in utilities will cause the export attribute to persist after the built-in has completed. If the assignment does not precede a utility name in the command, or if the assignment is a result of the operation of the *getopts* or *read* utilities, the export attribute will persist until the variable is unset.

–**b** Cause the shell to notify the user asynchronously of background job completions. The following message will be written to standard error:

```
"[%d]%c %s%s\n", <job-number>, <current>, <status>, <job-name>
```

where the fields are as follows:

<current> The character + identifies the job that would be used as a default for the *fg* or *bg* utilities; this job can also be specified using the *job_id* **%+** or **%%**.

The character – identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job_id* **%**–. For other jobs, this field is a space character. At most one job can be identified with + and at most one job can be identified with –. If there is any suspended job, then the current job will be a suspended job. If there are at least two suspended jobs, then the previous job will also be a suspended job.

<*job-number*> A number that can be used to identify the process group to the *wait*, *fg*, *bg* and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with %.

<*status*> Unspecified.

<*job-name*> Unspecified.

When the shell notifies the user a job has been completed, it may remove the job's process ID from the list of those known in the current shell execution environment; see Section 2.9.3 on page 50. Asynchronous notification will not be enabled by default.

–C (Upper-case C.) Prevent existing files from being overwritten by the shell's > redirection operator (see Section 2.7.2 on page 41); the >| redirection operator will override this *noclobber* option for an individual file.

–e When this option is on, if a simple command fails for any of the reasons listed in Section 2.8.1 on page 44 or returns an exit status value >0, and is not part of the compound list following a **while**, **until** or **if** keyword, and is not a part of an AND or OR list, and is not a pipeline preceded by the ! reserved word, then the shell will immediately exit.

–f The shell will disable pathname expansion.

EX **–h** Locate and remember utilities invoked by functions as those functions are defined (the utilities are normally located when the function is executed).

–m All jobs are run in their own process groups. Immediately before the shell issues a prompt after completion of the background job, a message reporting the exit status of the background job will be written to standard error. If a foreground job stops, the shell will write a message to standard error to that effect, formatted as described by the *jobs* utility. In addition, if a job changes status other than exiting (for example, if it stops for input or output or is stopped by a SIGSTOP signal), the shell will write a similar message immediately prior to writing the next prompt. This option is enabled by default for interactive shells.

–n The shell will read commands but not execute them; this can be used to check for shell script syntax errors. An interactive shell may ignore this option.

–o *option*
Set various options, many of which are equivalent to the single option letters. The following values of *option* are supported:

allexport Equivalent to **–a**.

errexit Equivalent to **–e**.

ignoreeof Prevent an interactive shell from exiting on end-of-file. This setting prevents accidental logouts when control-D is entered. A user must explicitly *exit* to leave the interactive shell.

monitor Equivalent to **–m**.

noclobber Equivalent to **–C** (upper-case C).

noglob Equivalent to **–f**.

noexec Equivalent to **–n**.

EX **nolog** Prevent the entry of function definitions into the command history. See **Command History List** on page 635.

notify Equivalent to **–b**.

nounset Equivalent to **–u**.

verbose Equivalent to **–v**.

vi Allow shell command-line editing using the built-in *vi* editor. Enabling **vi** mode disables any other command-line editing mode provided as an implementation extension.

It need not be possible to set **vi** mode on for certain block-mode terminals.

xtrace Equivalent to **–x**.

–u The shell will write a message to standard error when it tries to expand a variable that is not set and immediately exit. An interactive shell will not exit.

–v The shell will write its input to standard error as it is read.

–x The shell will write to standard error a trace for each command after it expands the command and before it executes it.

The default for all these options is off (unset) unless the shell was invoked with them on (see *sh*). All the positional parameters will be unset before any new values are assigned.

The remaining arguments will be assigned in order to the positional parameters. The special parameter # will be set to reflect the number of positional parameters.

The special argument **––** immediately following the *set* command name can be used to delimit the arguments if the first argument begins with + or –, or to prevent inadvertent listing of all shell variables when there are no arguments. The command *set* **––** without *argument* will unset all positional parameters and set the special parameter # to zero.

OB In the obsolescent version, the *set* command name followed by – with no other arguments will turn off the **–v** and **–x** options without changing the positional parameters. The *set* command name followed by – with other arguments will turn off the **–v** and **–x** options and assign the arguments to the positional parameters in order.

EXAMPLES

Write out all variables and their values:

```
set
```

Set $1, $2 and $3 and set $# to 3:

```
set c a b
```

Turn on the **–x** and **–v** options:

```
set –xv
```

Unset all positional parameters:

```
set --
```

Set $1 to the value of **x**, even if **x** begins with – or +:

```
set -- "$x"
```

Set the positional parameters to the expansion of **x**, even if **x** expands with a leading – or +:

```
set -- $x
```

EXIT STATUS

Zero.

2.14.12 shift — Shift Positional Parameters

SYNOPSIS

```
shift [n]
```

DESCRIPTION

The positional parameters will be shifted. Positional parameter 1 will be assigned the value of parameter (1+*n*), parameter 2 will be assigned the value of parameter (2+*n*), and so forth. The parameters represented by the numbers $# down to $#–*n*+1 will be unset, and the parameter # will be updated to reflect the new number of positional parameters.

The value *n* will be an unsigned decimal integer less than or equal to the value of the special parameter #. If *n* is not given, it will be assumed to be 1. If *n* is 0, the positional and special parameters will not be changed.

EXAMPLES

```
$ set a b c d e
$ shift 2
$ echo $*
c d e
```

EXIT STATUS

The exit status will be >0 if *n*>$#; otherwise, it will be zero.

2.14.13 times — Write Process Times

SYNOPSIS

```
times
```

DESCRIPTION

Write the accumulated user and system times for the shell and for all of its child processes, in the following POSIX Locale format:

```
"%dm%fs %dm%fs\n%dm%fs %dm%fs\n", <shell user minutes>,
<shell user seconds>, <shell system minutes>,
<shell system seconds>, <children user minutes>,
<children user seconds>, <children system minutes>
<children system seconds>
```

The four pairs of times correspond to the members of the **XSH** specification **<sys/times.h> tms** structure as returned by *times*( ): *tms_utime*, *tms_stime*, *tms_cutime* and *tms_cstime*, respectively.

EXAMPLES

```
$ times
0m0.43s 0m1.11s
8m44.18s 1m43.23s
```

EXIT STATUS

Zero.

2.14.14 trap — Trap Signals

SYNOPSIS

```
trap [action condition ...]
```

DESCRIPTION

If *action* is –, the shell will reset each *condition* to the default value. If *action* is null (''), the shell will ignore each specified *condition* if it arises. Otherwise, the argument *action* will be read and executed by the shell when one of the corresponding conditions arises. The action of the trap will override a previous action (either default action or one explicitly set). The value of $? after the trap action completes will be the value it had before the trap was invoked.

The condition can be EXIT, 0 (equivalent to EXIT) or a signal specified using a symbolic name, without the SIG prefix, as listed in the tables of signal names in the **XSH** specification under **<signal.h>**, for example, HUP, INT, QUIT, TERM. Implementations may permit lower-case signal names or names with the SIG prefix as an extension. Setting a trap for SIGKILL or SIGSTOP produces undefined results.

The environment in which the shell executes a trap on **EXIT** will be identical to the environment immediately after the last command executed before the trap on **EXIT** was taken.

Each time the trap is invoked, the *action* argument will be processed in a manner equivalent to:

```
eval "$action"
```

Signals that were ignored on entry to a non-interactive shell cannot be trapped or reset, although no error need be reported when attempting to do so. An interactive shell may reset or catch signals ignored on entry. Traps will remain in place for a given shell until explicitly changed with another *trap* command.

When a subshell is entered, traps are set to the default actions. This does not imply that the *trap* command cannot be used within the subshell to set new traps.

The *trap* command with no arguments will write to standard output a list of commands associated with each condition. The format is:

```
"trap -- %s %s...\n", <action>, <condition> ...
```

The shell will format the output, including the proper use of quoting, so that it is suitable for reinput to the shell as commands that achieve the same trapping results. For example:

```
save_traps=$(trap)
...
eval "$save_traps"
```

EX OB XSI-conformant systems also allow numeric signal numbers for the conditions, corresponding to the following signal names:

Signal Number	Signal Name
1	SIGHUP
2	SIGINT
3	SIGQUIT
6	SIGABRT
9	SIGKILL
14	SIGALRM
15	SIGTERM

The *trap* special built-in supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

EXAMPLES

Write out a list of all traps and actions:

```
trap
```

Set a trap so the *logout* utility in the *HOME* directory will execute when the shell terminates:

```
trap '$HOME/logout' EXIT
```

or

```
trap '$HOME/logout' 0
```

Unset traps on INT, QUIT, TERM and EXIT:

```
trap - INT QUIT TERM EXIT
```

EXIT STATUS

If the trap name or number is invalid, a non-zero exit status will be returned; otherwise, zero will be returned. For both interactive and non-interactive shells, invalid signal names or numbers will not be considered a syntax error and will not cause the shell to abort.

2.14.15 unset — Unset Values and Attributes of Variables and Functions

SYNOPSIS

```
unset [–fv] name ...
```

DESCRIPTION

Each variable or function specified by *name* will be unset.

If **–v** is specified, *name* refers to a variable name and the shell will unset it and remove it from the environment. Read-only variables cannot be unset.

If **–f** is specified, *name* refers to a function and the shell will unset the function definition.

If neither **–f** nor **–v** is specified, *name* refers to a variable; if a variable by that name does not exist, it is unspecified whether a function by that name, if any, will be unset.

Unsetting a variable or function that was not previously set is not considered an error and will not cause the shell to abort.

The *unset* special built-in supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

Note that:

```
VARIABLE=
```

is not equivalent to an *unset* of **VARIABLE**; in the example, **VARIABLE** is set to "". Also, the variables that can be *unset* should not be misinterpreted to include the special parameters (see Section 2.5.2 on page 27).

EXAMPLES

Unset *VISUAL* variable:

```
unset -v VISUAL
```

Unset the functions **foo** and **bar**:

```
unset –f foo bar
```

EXIT STATUS

0 All cases of *name* were successfully unset.
>0 At least one *name* could not be unset.

Chapter 3

Utilities

This chapter contains the definitions of the XSI utilities, as follows:

- mandatory utilities that are present on every XSI-conformant system
- optional utilities that may be present on XSI-conformant systems; optional utilities are marked (**OPTIONAL**) in their **NAME** lines
- software development utilities that are present only on systems supporting the software development facility; the development utilities are marked (**DEVELOPMENT**) in their **NAME** lines
- the FORTRAN utility that is present only on systems supporting the FORTRAN facility; the FORTRAN utility is marked (**FORTRAN**)
- utilities that are to be withdrawn and may be present on XSI-conformant systems; these utilities are marked (**TO BE WITHDRAWN**) in their **NAME** lines
- utilities that are withdrawn and may be present on XSI-conformant systems; these utilities are marked (**WITHDRAWN**) in their **NAME** lines.

NAME

admin – create and administer SCCS files (**DEVELOPMENT**)

SYNOPSIS

EX
```
admin -i[name][-n][-a login][-d flag][-f flag][-m mrlist]
[-r rel][-t[name][-y[comment]] newfile
admin -n[-a login][-d flag][-f flag][-m mrlist][-t[name]][-y[comment]]
newfile ...
admin [-a login][-d flag][-m mrlist][-r rel][-t[name]]
file ...
admin -h file ...
admin -z file ...
```

DESCRIPTION

The *admin* utility is used to create new SCCS files and change parameters of existing ones. If a named file does not exist, it is created, and its parameters are initialised according to the specified options. Parameters not initialised 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.

All SCCS filenames must be of the form s.*filename*. New SCCS filenames are given read-only permission mode. Write permission in the parent directory is required to create a file. All writing done by *admin* is to a temporary *x-file*, named x.*filename* (see *get*) created with read-only mode if *admin* is creating a new SCCS file, or created with the same mode as that of the SCCS file if the file already 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 occur.

The *admin* utility also uses a transient lock file (named z.*filename*), which is used to prevent simultaneous updates to the SCCS file. See *get* for further information.

OPTIONS

The *admin* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the **–i**, **–t** and **–y** options have optional option-arguments. These optional option-arguments cannot be presented as separate arguments. The following options are supported:

–n Create a new SCCS file. When **–n** is used without **–i**, the SCCS file is created with control information but without any file data.

–i[*name***]**
Specify 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 *name* option-argument is omitted, the text is obtained by reading the standard input. If this option is omitted, the SCCS file is created with control information but without any file data. The **–i** option implies the **–n** option.

–r *rel* Specify the *rel*ease into which the initial delta is inserted. 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***]**
Specify the *name* of a file from which descriptive text for the SCCS file is to be taken. In the case of existing SCCS files (neither **–i** nor **–n** is specified):

- A **–t** option without a *name* option-argument causes the removal of descriptive text (if any) currently in the SCCS file.

- A –**t** option with a *name* option-argument causes the text (if any) in the named file to replace the descriptive text (if any) currently in the SCCS file.

–**f** *flag* Specify 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 flags and their values are:

b Allow use of the –**b** option on a *get* command to create branch deltas.

c*ceil* Specify the highest release (that is, 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* Specify the lowest release (that is, 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* Specify the default delta number (SID) to be used by a *get* command.

i*str* Treat the ''No id keywords'' message issued by *get* or *delta* 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. If a value is supplied, the keywords must exactly match the given string; however, the string must contain a keyword, and no embedded newlines.

j Allow 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* Specify a *list* of releases to which deltas can no longer be made (that is, *get* –**e** against one of these locked releases fails). The *list* has the following syntax:

```
<list> ::= <range> | <list>, <range>
<range> ::= SID | a
```

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

n Cause *delta* to create a null delta in each of those releases (if any) being skipped when a delta is made in a new release (for example, 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* Substitute user definable *text* for all occurrences of the **%Q%** keyword in the SCCS file text retrieved by *get*.

m*mod* Specify the module name of the SCCS file substituted for all occurrences of the **%M%** keyword in the 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* Specify the *type* of module in the SCCS file substituted for all occurrences of **%Y%** keyword in the SCCS file text retrieved by *get*.

v*pgm* Cause *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 validation 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* Remove (delete) the specified *flag* from an SCCS file. 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*

Specify a *login* name, or numerical group ID, to be added to the list of users who 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 !, the users so specified are denied permission to make deltas.

–e *login*

Specify 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***]**

Insert the *comment* text into the SCCS file as a comment for the initial delta in a manner identical to that of *delta*. In the POSIX locale, omission of the **–y** option results in a default comment line being inserted in the form:

`"date and time created %s %s by %s", <`*date*`>, <`*time*`>, <`*login*`>`

where <*date*> is expressed in the *date* utility's %y/%m/%d format, <*time*> in the *date* utility's %T format and <*login*> is the login name of the user creating the file.

–m *mrlist*

Insert the list of modification request (MR) numbers 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 Check the structure of the SCCS file and compare the newly computed checksum (the sum of all the characters in the SCCS file except those in the first line) with the checksum that is stored in the first line of the SCCS file. Appropriate error diagnostics are produced.

–z Recompute the SCCS file checksum and store it 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.

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *admin* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

newfile A pathname of an SCCS file to be created.

If a single instance *file* or *newfile* is specified as –, 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.

STDIN

The standard input is a text file used only if the –**i** is specified without an option-argument or if a *file* or *newfile* operand is specified as –. If the first character of any standard input line is SOH (binary 001), the results are unspecified.

INPUT FILES

The existing SCCS files are text files of an unspecified format. The file named by the –**i** option's *name* option-argument is a text file; if the first character of any line in this file is SOH (binary 001), the results are unspecified.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *admin*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and the contents of the default –**y** comment.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Any SCCS files created are text files of an unspecified format. During processing of a *file*, a locking *z-file*, as described in *get*, may be created and deleted.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

It is recommended that directories containing SCCS files be writable by the owner only, and that SCCS files themselves be read-only. The mode of the directories should allow 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.

EXAMPLES

None.

FUTURE DIRECTIONS

A version of *admin* that fully supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** may be introduced in a future issue.

SEE ALSO

delta, *get*, *prs*, *what*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Conformance to Utility Syntax Guidelines mandated, with exceptions as noted.

Internationalised environment variable support mandated.

NAME

alias – Define or display aliases

SYNOPSIS

`alias [`*`alias-name`*`[=`*`string`*`] ...]`

DESCRIPTION

The *alias* utility creates or redefines alias definitions or writes the values of existing alias definitions to standard output. An alias definition provides a string value that replaces a command name when it is encountered. See Section 2.3.1 on page 24.

An alias definition affects the current shell execution environment and the execution environments of the subshells of the current shell. When used as specified by this document, the alias definition will not affect the parent process of the current shell nor any utility environment invoked by the shell. See Section 2.12 on page 63.

OPTIONS

None.

OPERANDS

The following operands are supported:

alias-name

Write the alias definition to standard output.

alias-name=*string*

Assign the value of *string* to the alias *alias-name*.

If no operands are given, all alias definitions will be written to standard output.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *alias*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The format for displaying aliases (when no operands or only *name* operands are specified) is:

```
"%s=%s\n", name, value
```

The *value* string will be written with appropriate quoting so that it is suitable for reinput to the shell. See the description of shell quoting in Section 2.2 on page 20.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 One of the *name* operands specified did not have an alias definition, or an error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES

1. Change *ls* to give a columnated, more annotated output:

   ```
   alias ls="ls –CF"
   ```

2. Create a simple ‘‘redo’’ command to repeat previous entries in the command history file:

   ```
   alias r='fc –s'
   ```

3. Use 1K units for *du*:

   ```
   alias du=du\ –k
   ```

4. Set up *nohup* so that it can deal with an argument that is itself an alias name:

   ```
   alias nohup="nohup "
   ```

FUTURE DIRECTIONS
None.

SEE ALSO
Section 2.9.5 on page 54.

CHANGE HISTORY
First released in Issue 4.

NAME

ar – create and maintain library archives

SYNOPSIS

EX `ar -d[-v][-l] archive file ...`

EX `ar -m[-abilv][posname] archive file ...`

UN EX `ar -p[-v][-s]archive [file ...]`

EX `ar -q[-clv] archive file ...`

EX `ar -r[-cuv][-abil][posname]archive file ...`

EX `ar -t[-v][-s]archive [file ...]`

EX `ar -x[-v][-sCT]archive [file ...]`

DESCRIPTION

The *ar* utility can be used to create and maintain groups of files combined into an archive. Once an archive has been created, new files can be added, and existing files can be extracted, deleted or replaced. When an archive consists entirely of valid object files, the implementation will format the archive so that it is usable as a library for link editing (see *c89*, *cc* and *fort77*). When some of the archived files are not valid object files, the suitability of the archive for library use is

EX undefined. If an archive file consists entirely of printable files, the entire archive file is printable.

EX 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* recognises as such in the archive, an archive symbol table is created in the archive file and maintained by *ar*; it is used by the link editor to search the archive file. Whenever the *ar* utility is used to create or update the contents of such an archive, the symbol table is rebuilt. The **–s** option forces the symbol table to be rebuilt.

All *file* operands can be pathnames. However, files within archives are named by a filename, which is the last component of the pathname used when the file was entered into the archive. The comparison of *file* operands to the names of files in archives is performed by comparing the last component of the operand to the name of the archive file.

It is unspecified whether multiple files in the archive may be identically named. In the case of

EX such files, however, each *file* and *posname* operand will match only the first archive file having a name that is the same as the last component of the operand.

OPTIONS

The *ar* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

EX **–a** Position new files in the archive after the file named by the *posname* operand.

EX **–b** Position new files in the archive before the file named by the *posname* operand.

–c Suppress the diagnostic message that is written to standard error by default when the archive file *archive* is created.

EX **–C** Prevent extracted files from replacing like-named files in the file system. This option is useful when **–T** is also used, to prevent truncated filenames from replacing files with the same prefix.

–d Delete one or more *file*s from *archive*.

EX **–i** Position new files in the archive before the file named by the *posname* operand (equivalent to **–b**).

EX **–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 (**TO BE WITHDRAWN**).

EX **–m** Move the named files. The **–a**, **–b** or **–i** options with the *posname* operand indicate the position; otherwise, move the files to the end of the archive.

–p Write the contents of the *file*s from *archive* to the standard output. If no *file*s are specified, the contents of all files in the archive will be written in the order of the archive.

EX **–q** Quickly append the named files to the end of the archive file. 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.

–r Replace or add *file*s to *archive*. If the archive named by *archive* does not exist, a new archive file will be created and a diagnostic message will be written to standard error (unless the **–c** option is specified). If no *file*s are specified and the *archive* exists, the results are undefined. Files that replace existing files will not change the order of the archive. Files that do not replace existing files will be appended to the archive.

UN **–s** Force the regeneration of the archive symbol table even if *ar* is not invoked with an option that will modify the archive file contents. This option is useful to restore the archive symbol table after it has been stripped; see *strip*.

–t Write a table of contents of *archive* to the standard output. The files specified by the *file* operands will be included in the written list. If no *file* operands are specified, all files in *archive* will be included in the order of the archive.

EX **–T** Allow filename truncation of extracted files whose archive names are longer than the file system can support. By default, extracting a file with a name that is too long is an error; a diagnostic message will be written and the file will not be extracted.

–u Update older files. When used with the **–r** option, files within the archive will be replaced only if the corresponding *file* has a modification time that is at least as new as the modification time of the file within the archive.

–v Give verbose output. When used with the option characters **–d**, **–r** or **–x**, write a detailed file-by-file description of the archive creation and maintenance activity, as described in **STDOUT**.

When used with **–p**, write the name of the file to the standard output before writing the file itself to the standard output, as described in **STDOUT**.

When used with **–t**, include a long listing of information about the files within the archive, as described in **STDOUT**.

–x Extract the files named by the *file* operands from *archive*. The contents of the archive file will not be changed. If no *file* operands are given, all files in the archive will be extracted. If the filename of a file extracted from the archive is longer than that supported in the directory to which it is being extracted, the results are undefined. The modification time of each file extracted will be set to the time the file is extracted from the archive.

OPERANDS

The following operands are supported:

archive A pathname of the archive file.

file A pathname. Only the last component will be used when comparing against the names of files in the archive. If two or more *file* operands have the same last pathname component (basename), the results are unspecified. The implementation's archive format will not truncate valid filenames of files added to or replaced in the archive.

EX *posname*

The name of a file in the archive file, used for relative positioning; see options –**m** and –**r**.

STDIN

Not used.

INPUT FILES

The input file named by *archive* must be a file in the format created by *ar* –**r**.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ar*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME

Determine the format and content for date and time strings written by *ar* –**tv**.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TMPDIR

Determine the pathname that overrides the default directory for temporary files, if any.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the –**d** option is used with the –**v** option, the standard output format is:

```
"d – %s\n", <file>
```

where <*file*> is the operand specified on the command line.

If the **–p** option is used with the **–v** option, *ar* will precede the contents of each file with:

```
"\n<%s>\n\n", <file>
```

where <*file*> is the operand specified on the command line, if *file* operands were specified, and the name of the file in the archive if they were not.

If the **–r** option is used with the **–v** option, and *file* is already in the archive, the standard output format is:

```
"r – %s\n", <file>
```

where <*file*> is the operand specified on the command line.

If *file* is being added to the archive with the **–r** option, the standard output format is:

```
"a – %s\n", <file>
```

where <*file*> is the operand specified on the command line.

If the **–t** option is used, *ar* writes the names of the files to the standard output in the format:

```
"%s\n", <file>
```

where <*file*> is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

If the **–t** option is used with the **–v** option, the standard output format is:

```
"%s %u/%u %u %s %d %d:%d %d %s\n", <member mode>, <user ID>,
<group ID>, <number of bytes in member>, <abbreviated month>,
<day-of-month>, <hour>, <minute>, <year>, <file>
```

Where:

<*file*> is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

<*member mode*>
: is formatted the same as the <*file mode*> string defined in the **STDOUT** section of *ls*, except that the first character, the <*entry type*>, is not used; the string represents the file mode of the archive member at the time it was added to or replaced in the archive.

The following represent the last-modification time of a file when it was most recently added to or replaced in the archive:

<*abbreviated month*>
: is equivalent to the **%b** format in *date*.

<*day-of-month*>
: is equivalent to the **%e** format in *date*.

<*hour*> is equivalent to the **%H** format in *date*.

<*minute*>
: is equivalent to the **%M** format in *date*.

<*year*> is equivalent to the **%Y** format in *date*.

When *LC_TIME* does not specify the POSIX locale, a different format and order of presentation of these fields relative to each other may be used in a format appropriate in the specified locale.

If the **–x** option is used with the **–v** option, the standard output format is:

```
"x – %s\n", <file>
```

where <*file*> is the operand specified on the command line, if *file* operands were specified, or the name of the file in the archive if they were not.

STDERR

Used only for diagnostic messages. The diagnostic message about creating a new archive when **–c** is not specified will not modify the exit status.

OUTPUT FILES

Archives are files with unspecified formats.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c89, *cc*, *cpio*, *pax*, *strip*, *tar*, the **XSH** specification **<unistd.h>** description of {POSIX_NO_TRUNC}.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard. The **–C** and **–T** options are added.

NAME

asa – interpret carriage-control characters

SYNOPSIS

```
asa [file ...]
```

DESCRIPTION

The *asa* utility will write its input files to standard output, mapping carriage-control characters from the text files to line-printer control sequences in an implementation-dependent manner.

The first character of every line will be removed from the input, and the following actions will be performed:

If the character removed is:

space The rest of the line will be output without change.

0 A newline character will be output, then the rest of the input line.

1 One or more implementation-dependent characters that causes an advance to the next page will be output, followed by the rest of the input line.

+ The newline character of the previous line will be replaced with one or more implementation-dependent characters that causes printing to return to column position 1, followed by the rest of the input line. If the + is the first character in the input, it will have the same effect as the space character.

The action of the *asa* utility is unspecified upon encountering any character other than those listed above as the first character in a line.

OPTIONS

None.

OPERANDS

file A pathname of a text file used for input. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *asa*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output will be the text from the input file modified as described in **DESCRIPTION**.

STDERR
None.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All input files were output successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
The following command:

```
asa file
```

permits the viewing of **file** (created by a program using FORTRAN-style carriage control characters) on a terminal.

The following command:

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

formats the FORTRAN output of **a.out** and directs it to the printer.

FUTURE DIRECTIONS
None.

SEE ALSO
fort77, *lp*.

CHANGE HISTORY
First released in Issue 4.

NAME

at – execute commands at a later time

SYNOPSIS

```
at [-m][-f file][-q queuename] -t time
at [-m][-f file][-q queuename] timespec ...
at -r at_job_id ...
at -l -q queuename
at -l [at_job_id ...]
```

DESCRIPTION

The *at* utility reads commands from standard input and groups them together as an *at-job*, to be executed at a later time.

The at-job will be executed in a separate invocation of the shell, running in a separate process group with no controlling terminal, except that the environment variables, current working directory, file creation mask and other implementation-dependent execution-time attributes in effect when the *at* utility is executed will be retained and used when the at-job is executed.

When the at-job is submitted, the *at_job_id* and scheduled time are written to standard error. The *at_job_id* is an identifier that will be a string consisting solely of alphanumeric characters and the period character. The *at_job_id* is assigned by the system when the job is scheduled such that it uniquely identifies a particular job.

User notification and the processing of the job's standard output and standard error are described under the –**m** option.

EX 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 a process with the appropriate privileges is allowed to submit a job. If only **at.deny** exists and is empty, global usage is permitted. The **at.allow** and **at.deny** files consist of one user name per line.

OPTIONS

The *at* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**f** *file* Specify the pathname of a file to be used as the source of the at-job, instead of standard input.

–**l** (The letter ell.) Report all jobs scheduled for the invoking user if no *at_job_id* operands are specified. If *at_job_id*s are specified, report only information for these jobs. The output will be written to standard output.

–**m** Send mail to the invoking user after the at-job has run, announcing its completion. Standard output and standard error produced by the at-job will be mailed to the user as well, unless redirected elsewhere. Mail will be sent even if the job produces no output.

If –**m** is not used, the job's standard output and standard error will be provided to the
EX user by means of mail, unless they are redirected elsewhere; if there is no such output to provide, the implementation need not notify the user of the job's completion.

–q *queuename*
Specify in which queue to schedule a job for submission. When used with the **–l** option, limit the search to that particular queue. By default, at-jobs will be scheduled in queue a. In contrast, queue b is reserved for batch jobs. (See the *batch* utility.) The meanings of all other *queuenames* are implementation-dependent.

–r Remove the jobs with the specified *at_job_id* operands that were previously scheduled by the *at* utility.

–t *time* Submit the job to be run at the time specified by the *time* option-argument, which must have the format as specified by the *touch* utility.

OPERANDS

The following operands are supported:

at_job_id
The name reported by a previous invocation of the *at* utility at the time the job was scheduled.

timespec Submit the job to be run at the date and time specified. All of the *timespec* operands are interpreted as if they were separated by space characters and concatenated, and are parsed as described in the grammar at the end of this section. The date and time are interpreted as being in the timezone of the user (as determined by the *TZ* variable), unless a timezone name appears as part of *time*, below.

In the POSIX locale, the following describes the three parts of the time specification string. All of the values from the LC_TIME categories in the POSIX locale are recognised in a case-insensitive manner.

time The *time* can be specified as one, two or four digits. One- and two-digit numbers are taken to be hours, four-digit numbers to be hours and minutes. The time can alternatively be specified as two numbers separated by a colon, meaning *hour*:*minute*. An AM/PM indication (one of the values from the **am_pm** keywords in the LC_TIME locale category) can follow the time; otherwise, a 24-hour clock time is understood. A timezone name can also follow to further qualify the time. The acceptable timezone names are implementation-dependent, except that they will be case-insensitive and the string **utc** is supported to indicate the time is in Coordinated Universal Time. The *time* field can also be one of the following tokens in the POSIX locale:

midnight
Indicates the time 12:00 am (00:00).

noon Indicates the time 12:00 pm.

now Indicate the current day and time. Invoking *at* <*now*> will submit an at-job for potentially immediate execution (that is, subject only to unspecified scheduling delays).

date An optional *date* can be specified as either a month name (one of the values from the **mon** or **abmon** keywords in the LC_TIME locale category) followed by a day number (and possibly year number preceded by a comma) or a day of the week (one of the values from the **day** or **abday** keywords in the LC_TIME locale category). Two special days are recognised in the POSIX locale:

today Indicates the current day.

tomorrow
Indicates the day following the current day.

If no *date* is given, **today** is assumed if the given time is greater than the current time, 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.

increment
The optional *increment* is a number preceded by a plus sign (+) and suffixed by one of the following: **minutes**, **hours**, **days**, **weeks**, **months** or **years**. (The singular forms will be also accepted.) The keyword **next** is equivalent to an increment number of +1. For example, the following are equivalent commands:

```
at 2pm + 1 week
at 2pm next week
```

The following grammar describes the precise format of *timespec* in the POSIX locale. The general conventions for this style of grammar are described in Section 1.8 on page 10. This formal syntax takes precedence over the preceding text syntax description. The longest possible token or delimiter will be recognised at a given point. When used in a *timespec*, white space also delimits tokens.

```
%token hr24clock_hr_min
%token hr24clock_hour
/*
  A hr24clock_hr_min is a one, two or four digit number.  A one or two
  digit number constitutes a hr24clock_hour.  A hr24clock_hour may be
  any of the single digits '0' - '9', or may be double digits, ranging
  from "00" - "23".  If a hr24clock_hr_min is a four digit number, the
  first two digits must be a valid hr24clock_hour, while the last two
  represent the number of minutes, from "00" - "59".
*/

%token wallclock_hr_min
%token wallclock_hour
/*
  A wallclock_hr_min is a one, two or four digit number.  A one or two
  digit number constitutes a wallclock_hour.  A wallclock_hour may be
  any of the single digits '1' - '9', or may be double digits, ranging
  from "01" - "12".  If a wallclock_hr_min is a four digit number, the
  first two digits must be a valid wallclock_hour, while the last two
  represent the number of minutes, from "00" - "59".
*/

%token minute
/*
  A minute is a one or two digit number whose values can be '0' - '9'
  or "00" - "59".
*/

%token day_number
/*
  A day_number is a number in the range appropriate for the particular
  month and year specified by month_name and year_number, respectively.
  If no year_number is given, the current year is assumed if the given
```

```
  date and time are later this year.  If no year_number is given and
  the date and time have already occurred this year and the month is
  not the current month, next year is the assumed year.
*/

%token year_number
/*
  A year_number is a four-digit number representing the year A.D., in
  which the at_job is to be run.
*/

%token inc_number
/*
  The inc_number is the number of times the succeeding increment
  period is to be added to the specified date and time.
*/

%token timezone_name
/*
  The name of an optional timezone suffix to the time field, in an
  implementation-dependent format.
*/

%token month_name
/*
  One of the values from the "mon" or "abmon" keywords in the LC_TIME
  locale category.
*/

%token day_of_week
/*
  One of the values from the "day" or "abday" keywords in the LC_TIME
  locale category.
*/

%token am_pm
/*
  One of the values from the "am_pm" keyword in the LC_TIME locale
  category.
*/

%start timespec
%%
timespec      : time
              | time date
              | time increment
              | time date increment
              | nowspec
              ;

nowspec       : "now"
              | "now" increment
              ;

time          : hr24clock_hr_min
              | hr24clock_hr_min timezone_name
              | hr24clock_hour ":" minute
```

```
                | hr24clock_hour ":" minute timezone_name
                | wallclock_hr_min am_pm
                | wallclock_hr_min am_pm timezone_name
                | wallclock_hour ":" minute am_pm
                | wallclock_hour ":" minute am_pm timezone_name
                | "noon"
                | "midnight"
                ;
date            : month_name day_number
                | month_name day_number "," year_number
                | day_of_week
                | "today"
                | "tomorrow"
                ;
increment       : "+" inc_number inc_period
                | "next" inc_period
                ;
inc_period      : "minute" | "minutes"
                | "hour" | "hours"
                | "day" | "days"
                | "week" | "weeks"
                | "month" | "months"
                | "year" | "years"
                ;
```

STDIN

The standard input must be a text file consisting of commands acceptable to the shell command language described in Chapter 2 on page 19. The standard input will only be used if no **–f** *file* option is specified.

INPUT FILES

See **STDIN**.

EX The text files **/usr/lib/cron/at.allow** and **/usr/lib/cron/at.deny** contain user names, one per line, of users who are, respectively, authorised or denied access to the *at* and *batch* utilities.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *at*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic

messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

LC_TIME
Determine the format and contents for date and time strings written and accepted by *at*.

SHELL Determine a name of a command interpreter to be used to invoke the at-job. If the variable is unset or null, *sh* will be used. If it is set to a value other than a name for *sh*, the implementation will do one of the following: use that shell; use *sh*; use the login shell from the user database; or any of the preceding accompanied by a warning diagnostic about which was chosen.

TZ Determine the timezone. The job will be submitted for execution at the time specified by *timespec* or **–t** *time* relative to the timezone specified by the *TZ* variable. If *timespec* specifies a timezone, it will override *TZ*. If *timespec* does not specify a timezone and *TZ* is unset or null, an unspecified default timezone will be used.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When standard input is a terminal, prompts of unspecified format for each line of the user input described in **STDIN** may be written to standard output.

In the POSIX locale, the following will be written to the standard output for each job when jobs are listed in response to the **–l** option:

```
"%s\t%s\n", at_job_id, <date>
```

where <*date*> is equivalent in format to the output of:

```
date +"%a %b %e %T %Y"
```

The date and time written will be adjusted so that they appear in the timezone of the user (as determined by the *TZ* variable).

STDERR
The following will be written to standard error when a job has been successfully submitted:

```
"job %s at %s\n", at_job_id, <date>
```

where <*date*> has the same format as is described in **STDOUT**. Neither this, nor warning messages concerning the selection of the command interpreter, are considered a diagnostic that changes the exit status.

Diagnostic messages, if any, are written to standard error.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The *at* utility successfully submitted, removed or listed a job or jobs.
>0 An error occurred.

CONSEQUENCES OF ERRORS

The job will not be scheduled, removed or listed.

APPLICATION USAGE

The format of the *at* command line shown here is guaranteed only for the POSIX locale. Other cultures may be supported with substantially different interfaces, although implementations are encouraged to provide comparable levels of functionality.

Since the commands run in a separate shell invocation, running in a separate process group with no controlling terminal, open file descriptors, traps and priority inherited from the invoking environment are lost.

Some implementations do not allow substitution of different shells using *SHELL*. System V systems, for example, have used the login shell value for the user in **/etc/passwd**. To select reliably another command interpreter, the user must include it as part of the script, such as:

```
$ at 1800
myshell myscript
job ... at ...
$
```

EXAMPLES

1. This sequence can be used at a terminal:

```
at –m 0730 tomorrow
sort < file >outfile
EOT
```

2. This sequence, which demonstrates redirecting standard error to a pipe, is useful in a command procedure (the sequence of output redirection specifications is significant):

```
at now + 1 hour <<!
diff file1 file2 2>&1 >outfile | mailx mygroup
!
```

3. To have a job reschedule itself, *at* can be invoked from within the at-job. For example, this daily processing script named **my.daily** will run every day (although *crontab* is a more appropriate vehicle for such work):

```
# my.daily runs every day
daily processing
at now tomorrow < my.daily
```

4. The spacing of the three portions of the POSIX locale *timespec* is quite flexible as long as there are no ambiguities. Examples of various times and operand presentation include:

```
at 0815am Jan 24
at 8 :15amjan24
at now "+ 1day"
at 5 pm FRIday
at ’17
       utc+
       30minutes’
```

FUTURE DIRECTIONS

None.

SEE ALSO

batch, *crontab*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

References to ‘‘superuser’’ have been changed to ‘‘process with appropriate privileges’’; otherwise, functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

awk – pattern scanning and processing language

SYNOPSIS

```
awk [-F ERE][-v assignment] ... program [argument ...]
awk [-F ERE] -v progfile] ... [-v assignment] ...[argument ...]
```

DESCRIPTION

The *awk* utility executes programs written in the *awk* programming language, which is specialised for textual 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 will be carried out.

Input is interpreted as a sequence of records. By default, a record is a line, but this can be changed by using the **RS** built-in variable. Each record of input is matched in turn against each pattern in the program. For each pattern matched, the associated action will be executed.

The *awk* utility interprets each input record as a sequence of fields where, by default, a field is a string of non-blank characters. This default white-space field delimiter can be changed by using the **FS** built-in variable or the **–F** *ERE*. The *awk* utility denotes the first field in a record $1, the second $2, and so forth. The symbol $0 refers to the entire record; setting any other field will cause the reevaluation of $0. Assigning to $0 will reset the values of all other fields and the **NF** built-in variable.

OPTIONS

The *awk* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–F *ERE*

Define the input field separator to be the extended regular expression *ERE*, before any input is read; see **Regular Expressions** on page 113.

–f *progfile*

Specifies the pathname of the file *progfile* containing an *awk* program. If multiple instances of this option are specified, the concatenation of the files specified as *progfile* in the order specified will be the *awk* program. The *awk* program can alternatively be specified in the command line as a single argument.

–v *assignment*

The *assignment* argument must be in the same form as an *assignment* operand. The specified variable assignment will occur prior to executing the *awk* program, including the actions associated with **BEGIN** patterns (if any). Multiple occurrences of this option can be specified.

OPERANDS

The following operands are supported:

program

If no **–f** option is specified, the first operand to *awk* will be the text of the *awk* program. The application will supply the *program* operand as a single argument to *awk*. If the text does not end in a newline character, *awk* will interpret the text as if it did.

argument

Either of the following two types of *argument* can be intermixed:

file A pathname of a file that contains the input to be read, which is matched against the set of patterns in the program. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

assignment

An operand that begins with an underscore or alphabetic character from the portable character set (see the table in the **XBD** specification, **Section 4.1**, **Portable Character Set**), followed by a sequence of underscores, digits and alphabetics from the portable character set, followed by the = character will specify a variable assignment rather than a pathname. The characters before the = represent the name of an *awk* variable; if that name is an *awk* reserved word (see **Grammar** on page 121) the behaviour is undefined. The characters following the equal sign will be interpreted as if they appeared in the *awk* program preceded and followed by a double-quote (") character, as a **STRING** token (see **Grammar** on page 121), except that if the last character is an unescaped backslash, it will be interpreted as a literal backslash rather than as the first character of the sequence \". The variable will be assigned the value of that **STRING** token. If that value is considered a *numeric string* (see **Expressions in awk** on page 107), the variable will also be assigned its numeric value. Each such variable assignment will occur just prior to the processing of the following *file*, if any. Thus, an assignment before the first *file* argument will be executed after the **BEGIN** actions (if any), while an assignment after the last *file* argument will occur before the **END** actions (if any). If there are no *file* arguments, assignments will be executed before processing the standard input.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

Input files to the *awk* program from any of the following sources:

- any *file* operands or their equivalents, achieved by modifying the *awk* variables **ARGV** and **ARGC**
- standard input in the absence of any *file* operands
- arguments to the **getline** function

must be text files. Whether the variable **RS** is set to a value other than a newline character or not, for these files, implementations support records terminated with the specified separator up to {LINE_MAX} bytes and may support longer records.

If **–f** *progfile* is specified, the files named by each of the *progfile* option-arguments must be text files containing an *awk* program.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *awk*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions and in comparisons of string values.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments and input files), the behaviour of character classes within regular expressions, the identification of characters as letters, and the mapping of upper- and lower-case characters for the **toupper** and **tolower** functions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_NUMERIC
Determine the radix character used when interpreting numeric input, performing conversions between numeric and string values and formatting numeric output. Regardless of locale, the period character (the decimal-point character of the POSIX locale) is the decimal-point character recognised in processing *awk* programs (including assignments in command-line arguments).

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path when looking for commands executed by *system*(*expr*), or input and output pipes. See the **XBD** specification, **Chapter 6**, **Environment Variables**.

In addition, all environment variables will be visible via the *awk* variable **ENVIRON**.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The nature of the output files depends on the *awk* program.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The nature of the output files depends on the *awk* program.

EXTENDED DESCRIPTION

Overall Program Structure

An *awk* program is composed of pairs of the form:

```
pattern { action }
```

Either the pattern or the action (including the enclosing brace characters) can be omitted.

A missing pattern will match any record of input, and a missing action will be equivalent to an action that writes the matched record of input to standard output.

Execution of the *awk* program starts by first executing the actions associated with all **BEGIN** patterns in the order they occur in the program. Then each *file* operand (or standard input if no files were specified) will be processed in turn by reading data from the file until a record separator is seen (a newline character by default), splitting the current record into fields using the current value of **FS** according to the rules in **Regular Expressions** on page 113, evaluating each pattern in the program in the order of occurrence, and executing the action associated with each pattern that matches the current record. The action for a matching pattern will be executed before evaluating subsequent patterns. Last, the actions associated with all **END** patterns will be executed in the order they occur in the program.

Expressions in awk

Expressions describe computations used in *patterns* and *actions*. In the following table, valid expression operations are given in groups from highest precedence first to lowest precedence last, with equal-precedence operators grouped between horizontal lines. In expression evaluation, where the grammar is formally ambiguous, higher precedence operators will be evaluated before lower precedence operators. In this table *expr*, *expr1*, *expr2* and *expr3* represent any expression, while *lvalue* represents any entity that can be assigned to (that is, on the left side of an assignment operator). The precise syntax of expressions is given in **Grammar** on page 121.

Syntax	Name	Type of Result	Associativity
(*expr*)	Grouping	type of *expr*	n/a
$*expr*	Field reference	string	n/a
++ *lvalue*	Pre-increment	numeric	n/a
–– *lvalue*	Pre-decrement	numeric	n/a
lvalue ++	Post-increment	numeric	n/a
lvalue ––	Post-decrement	numeric	n/a
expr ^ *expr*	Exponentiation	numeric	right
! *expr*	Logical not	numeric	n/a
+ *expr*	Unary plus	numeric	n/a
– *expr*	Unary minus	numeric	n/a
expr * *expr*	Multiplication	numeric	left
expr / *expr*	Division	numeric	left
expr % *expr*	Modulus	numeric	left
expr + *expr*	Addition	numeric	left
expr – *expr*	Subtraction	numeric	left
expr expr	String concatenation	string	left
expr < *expr*	Less than	numeric	none
expr <= *expr*	Less than or equal to	numeric	none
expr != *expr*	Not equal to	numeric	none
expr == *expr*	Equal to	numeric	none
expr > *expr*	Greater than	numeric	none
expr >= *expr*	Greater than or equal to	numeric	none
expr ~ *expr*	ERE match	numeric	none
expr !~ *expr*	ERE non-match	numeric	none
expr in array	Array membership	numeric	left
(*index*) in *array*	Multi-dimension array membership	numeric	left
expr && *expr*	Logical AND	numeric	left
expr | | *expr*	Logical OR	numeric	left
expr1 ? *expr2* : *expr3*	Conditional expression	type of selected *expr2* or *expr3*	right
lvalue ^= *expr*	Exponentiation assignment	numeric	right
lvalue %= *expr*	Modulus assignment	numeric	right
lvalue *= *expr*	Multiplication assignment	numeric	right
lvalue /= *expr*	Division assignment	numeric	right
lvalue += *expr*	Addition assignment	numeric	right
lvalue –= *expr*	Subtraction assignment	numeric	right
lvalue = *expr*	Assignment	type of *expr*	right

Table 3-1 Expressions in Decreasing Precedence in *awk*

Each expression has either a string value, a numeric value or both. Except as stated for specific contexts, the value of an expression will be implicitly converted to the type needed for the context in which it is used. A string value will be converted to a numeric value by the equivalent of the following calls to functions defined by the ISO C standard:

```
setlocale(LC_NUMERIC, "");
numeric_value = atof(string_value);
```

A numeric value that is exactly equal to the value of an integer will be converted to a string by the equivalent of a call to the **sprintf** function (see **String Functions** on page 118) with the string %d as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. Any other numeric value will be converted to a string by the equivalent of a call to the **sprintf** function with the value of the variable **CONVFMT** as the *fmt* argument and the numeric value being converted as the first and only *expr* argument. The result of the conversion is unspecified if the value of **CONVFMT** is not a floating-point format specification. This document specifies no explicit conversions between numbers and strings. An application can force an expression to be treated as a number by adding zero to it, or can force it to be treated as a string by concatenating the null string ("") to it.

A string value will be considered to be a *numeric string* in the following case:

1. Any leading and trailing blank characters will be ignored.
2. If the first unignored character is a + or –, it will be ignored.
3. If the remaining unignored characters would be lexically recognised as a **NUMBER** token (as described by the lexical conventions in **Grammar** on page 121), the string will be considered a *numeric string*.

If a – character is ignored in the above steps, the numeric value of the *numeric string* will be the negation of the numeric value of the recognised **NUMBER** token. Otherwise the numeric value of the *numeric string* will be the numeric value of the recognised **NUMBER** token. Whether or not a string is a *numeric string* will be relevant only in contexts where that term is used in this section.

When an expression is used in a Boolean context, if it has a numeric value, a value of zero is treated as false and any other value is treated as true. Otherwise, a string value of the null string is treated as false and any other value is treated as true. A Boolean context is one of the following:

- the first subexpression of a conditional expression
- an expression operated on by logical NOT, logical AND or logical OR
- the second expression of a **for** statement
- the expression of an **if** statement
- the expression of the **while** clause in either a **while** or **do...while** statement
- an expression used as a pattern (as in Overall Program Structure).

All arithmetic will follow the semantics of floating-point arithmetic as specified by the ISO C standard.

The value of the expression:

expr1 ^ *expr2*

will be equivalent to the value returned by the ISO C standard function call:

pow(*expr1*, *expr2*)

The expression:

lvalue ^= *expr*

will be equivalent to the ISO C standard expression:

lvalue = pow(*lvalue*, *expr*)

except that *lvalue* will be evaluated only once. The value of the expression:

expr1 % *expr2*

will be equivalent to the value returned by the ISO C standard function call:

fmod(*expr1*, *expr2*)

The expression:

lvalue %= *expr*

will be equivalent to the ISO C standard expression:

lvalue = fmod(*lvalue*, *expr*)

except that *lvalue* will be evaluated only once.

Variables and fields will be set by the assignment statement:

lvalue = *expression*

and the type of *expression* will determine the resulting variable type. The assignment includes the arithmetic assignments (**+=**, **–=**, ***=**, **/=**, **%=**, **^=**, **++**, **––**) all of which produce a numeric result. The left-hand side of an assignment and the target of increment and decrement operators can be one of a variable, an array with index or a field selector.

The *awk* language supplies arrays that are used for storing numbers or strings. Arrays need not be declared. They are initially empty, and their sizes will change dynamically. The subscripts, or element identifiers, are strings, providing a type of associative array capability. An array name followed by a subscript within square brackets can be used as an *lvalue* and thus as an expression, as described in the grammar (see **Grammar** on page 121). Unsubscripted array names can be used in only the following contexts:

- a parameter in a function definition or function call
- the **NAME** token following any use of the keyword **in** as specified in the grammar (see **Grammar** on page 121); if the name used in this context is not an array name, the behaviour is undefined.

A valid array *index* consists of one or more comma-separated expressions, similar to the way in which multi-dimensional arrays are indexed in some programming languages. Because *awk* arrays are really one dimensional, such a comma-separated list will be converted to a single string by concatenating the string values of the separate expressions, each separated from the other by the value of the **SUBSEP** variable. Thus, the following two index operations will be equivalent:

var[*expr1, expr2, ... exprn*]
var[*expr1* SUBSEP *expr2* SUBSEP ... SUBSEP *exprn*]

A multi-dimensioned *index* used with the **in** operator must be parenthesised. The **in** operator, which tests for the existence of a particular array element, will not cause that element to exist. Any other reference to a non-existent array element will automatically create it.

Comparisons (with the <, <=, !=, ==, > and >= operators) will be made numerically if both operands are numeric or if one is numeric and the other has a string value that is a numeric string. Otherwise, operands will be converted to strings as required and a string comparison will be made using the locale-specific collation sequence. The value of the comparison expression will be 1 if the relation is true, or 0 if the relation is false.

Variables and Special Variables

Variables can be used in an *awk* program by referencing them. With the exception of function parameters (see **User-defined Functions** on page 120), they are not explicitly declared. Uninitialised scalar variables and array elements have both a numeric value of zero and a string value of the empty string.

Field variables are designated by a $ followed by a number or numerical expression. The effect of the field number *expression* evaluating to anything other than a non-negative integer is unspecified; uninitialised variables or string values need not be converted to numeric values in this context. New field variables can be created by assigning a value to them. References to non-existent fields (that is, fields after **$NF**), will produce the null string. However, assigning to a non-existent field (for example, $(NF+2) = 5) will increase the value of **NF**, create any intervening fields with the null string as their values and cause the value of $0 to be recomputed, with the fields being separated by the value of **OFS**. Each field variable will have a string value when created. If the string, with any occurrence of the decimal-point character from the current locale changed to a period character, would be considered a *numeric string* (see **Expressions in awk** on page 107), the field variable will also have the numeric value of the *numeric string*.

Implementations support the following other special variables that are set by *awk*:

ARGC The number of elements in the **ARGV** array.

ARGV An array of command line arguments, excluding options and the *program* argument, numbered from zero to ARGC−1.

The arguments in **ARGV** can be modified or added to; **ARGC** can be altered. As each input file ends, *awk* will treat the next non-null element of **ARGV**, up to the current value of ARGC−1, inclusive, as the name of the next input file. Thus, setting an element of **ARGV** to null means that it will not be treated as an input file. The name – indicates the standard input. If an argument matches the format of an *assignment* operand, this argument will be treated as an assignment rather than a *file* argument.

CONVFMT

The **printf** format for converting numbers to strings (except for output statements, where **OFMT** is used); "%.6g" by default.

ENVIRON

The variable **ENVIRON** is an array representing the value of the environment, as described in the **XSH** specification under the *exec* functions. The indices of the array are strings consisting of the names of the environment variables, and the value of each array element is a string consisting of the value of that variable. If the value of an environment variable is considered a *numeric string* (see **Expressions in awk** on page 107), the array element will also have its numeric value.

In all cases where the behaviour of *awk* is affected by environment variables (including the environment of any commands that *awk* executes via the **system** function or via pipeline redirections with the **print** statement, the **printf** statement, or the **getline** function), the environment used will be the environment at the time *awk* began executing; it is implementation-dependent whether any modification of **ENVIRON** affects this environment.

FILENAME
A pathname of the current input file. Inside a **BEGIN** action the value is undefined. Inside an **END** action the value is the name of the last input file processed.

FNR The ordinal number of the current record in the current file. Inside a **BEGIN** action the value is zero. Inside an **END** action the value is the number of the last record processed in the last file processed.

FS Input field separator regular expression; a space character by default.

NF The number of fields in the current record. Inside a **BEGIN** action, the use of **NF** is undefined unless a **getline** function without a *var* argument is executed previously. Inside an **END** action, **NF** will retain the value it had for the last record read, unless a subsequent, redirected, **getline** function without a *var* argument is performed prior to entering the **END** action.

NR The ordinal number of the current record from the start of input. Inside a **BEGIN** action the value is zero. Inside an **END** action the value is the number of the last record processed.

OFMT The **printf** format for converting numbers to strings in output statements (see **Output Statements** on page 116); "%.6g" by default. The result of the conversion is unspecified if the value of **OFMT** is not a floating-point format specification.

OFS The **print** statement output field separation; a space character by default.

ORS The **print** statement output record separator; a newline character by default.

RLENGTH
The length of the string matched by the **match** function.

RS The first character of the string value of **RS** is the input record separator; a newline character by default. If **RS** contains more than one character, the results are unspecified. If **RS** is null, then records are separated by sequences of one or more blank lines, leading or trailing blank lines do not result in empty records at the beginning or end of the input, and a newline character is always a field separator, no matter what the value of **FS** is.

RSTART
The starting position of the string matched by the **match** function, numbering from 1. This is always equivalent to the return value of the **match** function.

SUBSEP
The subscript separator string for multi-dimensional arrays; the default value is implementation-dependent.

Regular Expressions

The *awk* utility makes use of the extended regular expression notation (see the **XBD** specification, **Section 7.4**, **Extended Regular Expressions**) except that it will allow the use of C-language conventions for escaping special characters within the EREs, as specified in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v) and the following table; these escape sequences will be recognised both inside and outside bracket expressions. Note that records need not be separated by newline characters and string constants can contain newline characters, so even the \n sequence is valid in *awk* EREs. Using a slash character within the regular expression requires the escaping shown in the following table:

Escape Sequence	Description	Meaning
\"	Backslash quotation-mark	Quotation-mark character
\/	Backslash slash	Slash character
ddd	A backslash character followed by the longest sequence of one, two or three octal-digit characters (01234567). If all of the digits are 0, (that is, representation of the NUL character), the behaviour is undefined.	The character whose encoding is represented by the one-, two- or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-dependent. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
c	A backslash character followed by any character not described in this table or in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v)	Undefined

Table 3-2 Escape Sequences in *awk*

A regular expression can be matched against a specific field or string by using one of the two regular expression matching operators, ~ and !~. These operators interpret their right-hand operand as a regular expression and their left-hand operand as a string. If the regular expression matches the string, the ~ expression will evaluate to a value of 1, and the !~ expression will evaluate to a value of 0. (The regular expression matching operation is as defined by the term matched in the **XBD** specification, **Section 7.1**, **Regular Expression Definitions**, where a match occurs on any part of the string unless the regular expression is limited with the circumflex or dollar sign special characters.) If the regular expression does not match the string, the ~ expression will evaluate to a value of 0, and the !~ expression will evaluate to a value of 1. If the right-hand operand is any expression other than the lexical token **ERE**, the string value of the expression will be interpreted as an extended regular expression, including the escape conventions described above. Note that these same escape conventions also will be applied in the determining the value of a string literal (the lexical token **STRING**), and thus will be applied a second time when a string literal is used in this context.

When an **ERE** token appears as an expression in any context other than as the right-hand of the ~ or !~ operator or as one of the built-in function arguments described below, the value of the resulting expression will be the equivalent of:

```
$0 ~ /ere/
```

The *ere* argument to the **gsub**, **match**, **sub** functions, and the *fs* argument to the **split** function (see **String Functions** on page 118) will be interpreted as extended regular expressions. These can be either **ERE** tokens or arbitrary expressions, and will be interpreted in the same manner as the right-hand side of the ~ or !~ operator.

An extended regular expression can be used to separate fields by using the **–F** *ERE* option or by assigning a string containing the expression to the built-in variable **FS**. The default value of the **FS** variable will be a single space character. The following describes **FS** behaviour:

1. If **FS** is a single character:
   a. If **FS** is the space character, skip leading and trailing blank characters; fields will be delimited by sets of one or more blank characters.
   b. Otherwise, if **FS** is any other character *c*, fields will be delimited by each single occurrence of *c*.
2. Otherwise, the string value of **FS** will be considered to be an extended regular expression. Each occurrence of a sequence matching the extended regular expression will delimit fields.

Except in the **gsub**, **match**, **split** and **sub** built-in functions, regular expression matching will be based on input records; that is, record separator characters (the first character of the value of the variable **RS**, a newline character by default) cannot be embedded in the expression, and no expression will match the record separator character. If the record separator is not a newline character, newline characters embedded in the expression can be matched. In those four built-in functions, regular expression matching will be based on text strings; that is, any character (including the newline character and the record separator) can be embedded in the pattern and an appropriate pattern will match any character. However, in all *awk* regular expression matching, the use of one or more NUL characters in the pattern, input record or text string produces undefined results.

Patterns

A *pattern* is any valid *expression*, a range specified by two expressions separated by comma, or one of the two special patterns **BEGIN** or **END**.

Special Patterns

The *awk* utility recognises two special patterns, **BEGIN** and **END**. Each **BEGIN** pattern will be matched once and its associated action executed before the first record of input is read (except possibly by use of the **getline** function (see **Input/Output and General Functions** on page 119) in a prior **BEGIN** action) and before command line assignment is done. Each **END** pattern will be matched once and its associated action executed after the last record of input has been read. These two patterns will have associated actions.

BEGIN and **END** will not combine with other patterns. Multiple **BEGIN** and **END** patterns are allowed. The actions associated with the **BEGIN** patterns will be executed in the order specified in the program, as are the **END** actions. An **END** pattern can precede a **BEGIN** pattern in a program.

If an *awk* program consists of only actions with the pattern **BEGIN**, and the **BEGIN** action contains no **getline** function, *awk* will exit without reading its input when the last statement in the last **BEGIN** action is executed. If an *awk* program consists of only actions with the pattern **END** or only actions with the patterns **BEGIN** and **END**, the input will be read before the statements in the **END** actions are executed.

Expression Patterns

An expression pattern will be evaluated as if it were an expression in a Boolean context. If the result is true, the pattern will be considered to match, and the associated action (if any) will be executed. If the result is false, the action will not be executed.

Pattern Ranges

A pattern range consists of two expressions separated by a comma; in this case, the action will be performed for all records between a match of the first expression and the following match of the second expression, inclusive. At this point, the pattern range can be repeated starting at input records subsequent to the end of the matched range.

Actions

An action is a sequence of statements as shown in the grammar in **Grammar** on page 121. Any single statement can be replaced by a statement list enclosed in braces. The statements in a statement list must be separated by newline characters or semicolons, and will be executed sequentially in the order that they appear.

The *expression* acting as the conditional in an **if** statement will be evaluated and if it is non-zero or non-null, the following *statement* will be executed; otherwise, if **else** is present, the statement following the **else** will be executed.

The **if**, **while**, **do** ... **while**, **for**, **break** and **continue** statements are based on the ISO C standard, except that the Boolean expressions are treated as described in **Expressions in awk** on page 107, and except in the case of:

```
for (variable in array)
```

which will iterate, assigning each *index* of *array* to *variable* in an unspecified order. The results of adding new elements to *array* within such a **for** loop are undefined. If a **break** or **continue** statement occurs outside of a loop, the behaviour is undefined.

The **delete** statement will remove an individual array element. Thus, the following code will delete an entire array:

```
for (index in array)
       delete array[index]
```

The **next** statement will cause all further processing of the current input record to be abandoned. The behaviour is undefined if a **next** statement appears or is invoked in a **BEGIN** or **END** action.

The **exit** statement will invoke all **END** actions in the order in which they occur in the program source and then terminate the program without reading further input. An **exit** statement inside an **END** action will terminate the program without further execution of **END** actions. If an expression is specified in an **exit** statement, its numeric value will be the exit status of *awk*, unless subsequent errors are encountered or a subsequent **exit** statement with an expression is executed.

Output Statements

Both **print** and **printf** statements write to standard output by default. The output is written to the location specified by *output_redirection* if one is supplied, as follows:

```
> expression
>> expression
| expression
```

In all cases, the *expression* will be evaluated to produce a string that is used as a full pathname to write into (for > or >>) or as a command to be executed (for |). Using the first two forms, if the file of that name is not currently open, it will be opened, creating it if necessary and using the first form, truncating the file. The output then will be appended to the file. As long as the file remains open, subsequent calls in which *expression* evaluates to the same string value simply will append output to the file. The file remains open until the **close** function (see **Input/Output and General Functions** on page 119) is called with an expression that evaluates to the same string value.

The third form will write output onto a stream piped to the input of a command. The stream will be created if no stream is currently open with the value of *expression* as its command name. The stream created will be equivalent to one created by a call to the **XSH** specification *popen*( ) function with the value of *expression* as the *command* argument and a value of **w** as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value will write output to the existing stream. The stream will remain open until the **close** function (see **Input/Output and General Functions** on page 119) is called with an expression that evaluates to the same string value. At that time, the stream will be closed as if by a call to the **XSH** specification *pclose*( ) function.

As described in detail by the grammar in **Grammar** on page 121, these output statements take a comma-separated list of *expression*s referred in the grammar by the non-terminal symbols **expr_list**, **print_expr_list** or **print_expr_list_opt**. This list is referred to here as the *expression list*, and each member is referred to as an *expression argument*.

The **print** statement will write the value of each expression argument onto the indicated output stream separated by the current output field separator (see variable **OFS** above), and terminated by the output record separator (see variable **ORS** above). All expression arguments will be taken as strings, being converted if necessary; this conversion will be as described in **Expressions in awk** on page 107, with the exception that the **printf** format in **OFMT** will be used instead of the value in **CONVFMT**. An empty expression list will stand for the whole input record ($0).

The **printf** statement will produce output based on a notation similar to the File Format Notation used to describe file formats in this document (see the **XBD** specification, **Chapter 3**, **File Format Notation**). Output will be produced as specified with the first expression argument as the string <*format*> and subsequent expression arguments as the strings <*arg1*> to <*argn*>, inclusive, with the following exceptions:

1. The *format* will be an actual character string rather than a graphical representation. Therefore, it cannot contain empty character positions. The space character in the *format* string, in any context other than a *flag* of a conversion specification, will be treated as an ordinary character that is copied to the output.
2. If the character set contains a Δ character and that character appears in the *format* string, it will be treated as an ordinary character that is copied to the output.
3. The *escape sequences* beginning with a backslash character will be treated as sequences of ordinary characters that are copied to the output. Note that these same sequences will be

interpreted lexically by *awk* when they appear in literal strings, but they will not be treated specially by the **printf** statement.

4. A *field width* or *precision* can be specified as the * character instead of a digit string. In this case the next argument from the expression list will be fetched and its numeric value taken as the field width or precision.
5. The implementation will not precede or follow output from the d or u conversion specifications with blank characters not specified by the *format* string.
6. The implementation will not precede output from the o conversion specification with leading zeros not specified by the *format* string.
7. For the c conversion specification: if the argument has a numeric value, the character whose encoding is that value will be output. If the value is zero or is not the encoding of any character in the character set, the behaviour is undefined. If the argument does not have a numeric value, the first character of the string value will be output; if the string does not contain any characters the behaviour is undefined.
8. For each conversion specification that consumes an argument, the next expression argument will be evaluated. With the exception of the c conversion, the value will be converted (according to the rules specified in **Expressions in awk** on page 107) to the appropriate type for the conversion specification.
9. If there are insufficient expression arguments to satisfy all the conversion specifications in the *format* string, the behaviour is undefined.
10. If any character sequence in the *format* string begins with a % character, but does not form a valid conversion specification, the behaviour is unspecified.

Both **print** and **printf** can output at least {LINE_MAX} bytes.

Functions

The *awk* language has a variety of built-in functions: arithmetic, string, input/output and general.

Arithmetic Functions

The arithmetic functions, except for **int**, are based on the ISO C standard. The behaviour is undefined in cases where the ISO C standard specifies that an error be returned or that the behaviour is undefined. Although the grammar (see **Grammar** on page 121) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the **[ ]** brackets), such use is undefined.

`atan2(`*y*`,`*x*`)`
: Return arctangent of *y* /*x*.

`cos(`*x*`)` Return cosine of *x*, where *x* is in radians.

`sin(`*x*`)` Return sine of *x*, where *x* is in radians.

`exp(`*x*`)` Return the exponential function of *x*.

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

`sqrt(`*x*`)`
: Return the square root of *x*.

`int(`*x*`)` Truncate its argument to an integer. It will be truncated toward 0 when $x > 0$.

`rand( )` Return a random number *n*, such that $0 \le n < 1$.

`srand(`**[**`expr`**]**`)`
Set the seed value for **rand** to *expr* or use the time of day if *expr* is omitted. The previous seed value will be returned.

String Functions

The string functions in the following list shall be supported. Although the grammar (see **Grammar** on page 121) permits built-in functions to appear with no arguments or parentheses, unless the argument or parentheses are indicated as optional in the following list (by displaying them within the **[ ]** brackets), such use is undefined.

`gsub(`*ere*`, `*repl*`[,`*in*`])`
Behave like **sub** (see below), except that it will replace all occurrences of the regular expression (like the *ed* utility global substitute) in $0 or in the *in* argument, when specified.

`index(`*s*`, `*t*`)`
Return the position, in characters, numbering from 1, in string *s* where string *t* first occurs, or zero if it does not occur at all.

`length[([`*s*`])]`
Return the length, in characters, of its argument taken as a string, or of the whole
EX record, $0, if there is no argument. The use of no argument and no parentheses with **length** is obsolescent in the ISO/IEC 9945-2:1993 standard; to be fully portable to POSIX systems, the application must use length($0) for the length of the whole record. However, X/Open systems will continue to support this usage indefinitely.

`match(`*s*`, `*ere*`)`
Return the position, in characters, numbering from 1, in string *s* where the extended regular expression *ere* occurs, or zero if it does not occur at all. RSTART will be set to the starting position (which is the same as the returned value), zero if no match is found; RLENGTH will be set to the length of the matched string, –1 if no match is found.

`split(`*s*`, `*a*`[,`*fs*`])`
Split the string *s* into array elements *a*[1], *a*[2], ..., *a*[*n*], and return *n*. The separation will be done with the extended regular expression *fs* or with the field separator **FS** if *fs* is not given. Each array element will have a string value when created. If the string assigned to any array element, with any occurrence of the decimal-point character from the current locale changed to a period character, would be considered a *numeric string* (see **Expressions in awk** on page 107), the array element will also have the numeric value of the *numeric string*. The effect of a null string as the value of *fs* is unspecified.

`sprintf(`*fmt*`, `*expr*`, `*expr*`, ...)`
Format the expressions according to the **printf** format given by *fmt* and return the resulting string.

`sub(`*ere*`, `*repl*`[,`*in*`])`
Substitute the string *repl* in place of the first instance of the extended regular expression *ERE* in string *in* and return the number of substitutions. An ampersand (&) appearing in the string *repl* will be replaced by the string from *in* that matches the regular expression. For each occurrence of backslash (\) encountered when scanning the string *repl* from beginning to end, the next character is taken literally and loses its special

meaning (for example, \& will be interpreted as a literal ampersand character). Except for & and \, it is unspecified what the special meaning of any such character is. If *in* is specified and it is not an *lvalue* (see **Expressions in awk** on page 107), the behaviour is undefined. If *in* is omitted, *awk* will substitute in the current record ($0).

`substr(`*s*`, `*m*`[,`*n*`])`
: Return the at most *n*-character substring of *s* that begins at position *m*, numbering from 1. If *n* is missing, the length of the substring will be limited by the length of the string *s*.

`tolower(`*s*`)`
: Return a string based on the string *s*. Each character in *s* that is an upper-case letter specified to have a **tolower** mapping by the LC_CTYPE category of the current locale will be replaced in the returned string by the lower-case letter specified by the mapping. Other characters in *s* will be unchanged in the returned string.

`toupper(`*s*`)`
: Return a string based on the string *s*. Each character in *s* that is a lower-case letter specified to have a **toupper** mapping by the LC_CTYPE category of the current locale will be replaced in the returned string by the upper-case letter specified by the mapping. Other characters in *s* will be unchanged in the returned string.

All of the preceding functions that take *ERE* as a parameter expect a pattern or a string valued expression that is a regular expression as defined in **Regular Expressions** on page 113.

Input/Output and General Functions

The input/output and general functions are:

`close(`*expression*`)`
: Close the file or pipe opened by a **print** or **printf** statement or a call to **getline** with the same string-valued *expression*. The limit on the number of open *expression* arguments is implementation-dependent. If the close was successful, the function will return zero; otherwise, it will return non-zero.

`expression | getline [`*var*`]`
: Read a record of input from a stream piped from the output of a command. The stream will be created if no stream is currently open with the value of *expression* as its command name. The stream created will be equivalent to one created by a call to the *popen*( ) function with the value of *expression* as the *command* argument and a value of **r** as the *mode* argument. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value will read subsequent records from the file. The stream will remain open until the **close** function is called with an expression that evaluates to the same string value. At that time, the stream will be closed as if by a call to the *pclose*( ) function. If *var* is missing, $0 and **NF** will be set; otherwise, *var* will be set.

  The **getline** operator can form ambiguous constructs when there are unparenthesised operators (including concatenate) to the left of the | (to the beginning of the expression containing **getline**). In the context of the $ operator, | behaves as if it had a lower precedence than $. The result of evaluating other operators is unspecified, and portable applications must parenthesise properly all such usages.

`getline`
: Set $0 to the next input record from the current input file. This form of **getline** will set the **NF**, **NR** and **FNR** variables.

`getline` *var*

Set variable *var* to the next input record from the current input file. This form of **getline** will set the **FNR** and **NR** variables.

`getline [`*var*`] <` *expression*

Read the next record of input from a named file. The *expression* will be evaluated to produce a string that is used as a full pathname. If the file of that name is not currently open, it will be opened. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value will read subsequent records from the file. The file will remain open until the **close** function is called with an expression that evaluates to the same string value. If *var* is missing, $0 and **NF** will be set; otherwise, *var* will be set.

The **getline** operator can form ambiguous constructs when there are unparenthesised binary operators (including concatenate) to the right of the < (up to the end of the expression containing the **getline**). The result of evaluating such a construct is unspecified, and portable applications must parenthesise properly all such usages.

`system(`*expression*`)`

Execute the command given by *expression* in a manner equivalent to the **XSH** specification *system*( ) function and return the exit status of the command.

All forms of **getline** will return 1 for successful input, zero for end of file, and –1 for an error.

Where strings are used as the name of a file or pipeline, the strings must be textually identical. The terminology ‘‘same string value’’ implies that ‘‘equivalent strings’’, even those that differ only by space characters, represent different files.

User-defined Functions

The *awk* language also provides user-defined functions. Such functions can be defined as:

```
function name(args, ...) { statements }
```

A function can be referred to anywhere in an *awk* program; in particular, its use can precede its definition. The scope of a function will be global.

Function arguments can be either scalars or arrays; the behaviour is undefined if an array name is passed as an argument that the function uses as a scalar, or if a scalar expression is passed as an argument that the function uses as an array. Function arguments will be passed by value if scalar and by reference if array name. Argument names will be local to the function; all other variable names will be global. The same name will not be used as both an argument name and as the name of a function or a special *awk* variable. The same name must not be used both as a variable name with global scope and as the name of a function. The same name must not be used within the same scope both as a scalar variable and as an array.

The number of parameters in the function definition need not match the number of parameters in the function call. Excess formal parameters can be used as local variables. If fewer arguments are supplied in a function call than are in the function definition, the extra parameters that are used in the function body as scalars will be initialised with a string value of the null string and a numeric value of zero, and the extra parameters that are used in the function body as arrays will be initialised as empty arrays. If more arguments are supplied in a function call than are in the function definition, the behaviour is undefined.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. Function calls can be nested and recursive calls can be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function’s parameters will be unchanged, except for array parameters passed by reference. The

return statement can be used to return a value. If a **return** statement appears outside of a function definition, the behaviour is undefined.

In the function definition, newline characters are optional before the opening brace and after the closing brace. Function definitions can appear anywhere in the program where a *pattern-action* pair is allowed.

Grammar

The grammar in this section and the lexical conventions in the following section will together describe the syntax for *awk* programs. The general conventions for this style of grammar are described in Section 1.8 on page 10. A valid program can be represented as the non-terminal symbol *program* in the grammar. This formal syntax takes precedence over the preceding text syntax description.

```
%token NAME NUMBER STRING ERE
%token FUNC_NAME            /* name followed by '(' without white space */

/* Keywords  */
%token           Begin   End
/*              'BEGIN' 'END'                                          */

%token           Break   Continue   Delete   Do   Else
/*              'break' 'continue' 'delete' 'do' 'else'                */

%token           Exit   For   Function   If   In
/*              'exit' 'for' 'function' 'if' 'in'                      */

%token           Next   Print   Printf   Return   While
/*              'next' 'print' 'printf' 'return' 'while'               */

/* Reserved function names */
%token BUILTIN_FUNC_NAME     /* one token for the following:
                              * atan2 cos sin exp log sqrt int rand srand
                              * gsub index length match split sprintf sub
                              * substr tolower toupper close system
                              */
%token GETLINE              /* Syntactically different from other built-ins */

/* Two-character tokens */
%token  ADD_ASSIGN SUB_ASSIGN MUL_ASSIGN DIV_ASSIGN MOD_ASSIGN POW_ASSIGN
/*         '+='       '-='       '*='       '/='       '%='      '^='       */

%token  OR   AND  NO_MATCH  EQ   LE   GE   NE  INCR  DECR  APPEND
/*      '||'  '&&'   '!~'   '==' '<=' '>=' '!=' '++'  '--'   '>>'   */

/* One-character tokens */
%token '{' '}' '(' ')' '[' ']' ',' ';' NEWLINE
%token '+' '-' '*' '%' '^' '!' '>' '<' '|' '?' ':' '~' '$' '='

%start program
%%

program              : item_list
                     | actionless_item_list
                     ;

item_list            : newline_opt
                     | actionless_item_list  item terminator
                     | item_list             item terminator
                     | item_list           action terminator
                     ;
```

```
actionless_item_list  : item_list            pattern terminator
                      | actionless_item_list pattern terminator
                      ;
item                  : pattern action
                      | Function NAME      '(' param_list_opt ')'
                            newline_opt action
                      | Function FUNC_NAME '(' param_list_opt ')'
                            newline_opt action
                      ;
param_list_opt        : /* empty */
                      | param_list
                      ;
param_list            : NAME
                      | param_list ',' NAME
                      ;
pattern               : Begin
                      | End
                      | expr
                      | expr ',' newline_opt expr
                      ;
action                : '{' newline_opt                              '}'
                      | '{' newline_opt terminated_statement_list    '}'
                      | '{' newline_opt unterminated_statement_list  '}'
                      ;
terminator            : terminator ';'
                      | terminator NEWLINE
                      |            ';'
                      |            NEWLINE
                      ;
terminated_statement_list:  terminated_statement
                      | terminated_statement_list  terminated_statement
                      ;
unterminated_statement_list:  unterminated_statement
                      | terminated_statement_list  unterminated_statement
                      ;
terminated_statement  : action newline_opt
                      | If '(' expr ')' newline_opt terminated_statement
                      | If '(' expr ')' newline_opt terminated_statement
                          Else newline_opt terminated_statement
                      | While '(' expr ')' newline_opt terminated_statement
                      | For '(' simple_statement_opt ';'
                           expr_opt ';' simple_statement_opt ')' newline_opt
                          terminated_statement
                      | For '(' NAME In NAME ')' newline_opt
                           terminated_statement
                      | ';' newline_opt
                      | terminatable_statement  NEWLINE newline_opt
                      | terminatable_statement  ';'     newline_opt
                      ;
unterminated_statement:  terminatable_statement
                      | If '(' expr ')' newline_opt unterminated_statement
                      | If '(' expr ')' newline_opt terminated_statement
                          Else newline_opt unterminated_statement
                      | While '(' expr ')' newline_opt unterminated_statement
```

```
                        | For '(' simple_statement_opt ';'
                              expr_opt ';' simple_statement_opt ')' newline_opt
                              unterminated_statement
                        | For '(' NAME In NAME ')' newline_opt
                              unterminated_statement
                        ;
terminatable_statement: simple_statement
                        | Break
                        | Continue
                        | Next
                        | Exit expr_opt
                        | Return expr_opt
                        | Do newline_opt terminated_statement While '(' expr ')'
                        ;
simple_statement_opt  : /* empty */
                        | simple_statement
                        ;
simple_statement      : Delete NAME '[' expr_list ']'
                        | expr
                        | print_statement
                        ;
print_statement       : simple_print_statement
                        | simple_print_statement output_redirection
                        ;
simple_print_statement : Print  print_expr_list_opt
                        | Print  '(' multiple_expr_list ')'
                        | Printf print_expr_list
                        | Printf '(' multiple_expr_list ')'
                        ;
output_redirection    : '>'    expr
                        | APPEND expr
                        | '|'    expr
                        ;
expr_list_opt         : /* empty */
                        | expr_list
                        ;
expr_list             : expr
                        | multiple_expr_list
                        ;
multiple_expr_list    : expr ',' newline_opt expr
                        | multiple_expr_list ',' newline_opt expr
                        ;
expr_opt              : /* empty */
                        | expr
                        ;
expr                  : unary_expr
                        | non_unary_expr
                        ;
unary_expr            : '+' expr
                        | '-' expr
                        | unary_expr '^'      expr
                        | unary_expr '*'      expr
                        | unary_expr '/'      expr
```

```
                        | unary_expr '%'      expr
                        | unary_expr '+'      expr
                        | unary_expr '-'      expr
                        | unary_expr          non_unary_expr
                        | unary_expr '<'      expr
                        | unary_expr LE       expr
                        | unary_expr NE       expr
                        | unary_expr EQ       expr
                        | unary_expr '>'      expr
                        | unary_expr GE       expr
                        | unary_expr '~'      expr
                        | unary_expr NO_MATCH expr
                        | unary_expr In NAME
                        | unary_expr AND newline_opt expr
                        | unary_expr OR  newline_opt expr
                        | unary_expr '?' expr ':' expr
                        | unary_input_function
                        ;
non_unary_expr          : '(' expr ')'
                        | '!' expr
                        | non_unary_expr '^'      expr
                        | non_unary_expr '*'      expr
                        | non_unary_expr '/'      expr
                        | non_unary_expr '%'      expr
                        | non_unary_expr '+'      expr
                        | non_unary_expr '-'      expr
                        | non_unary_expr          non_unary_expr
                        | non_unary_expr '<'      expr
                        | non_unary_expr LE       expr
                        | non_unary_expr NE       expr
                        | non_unary_expr EQ       expr
                        | non_unary_expr '>'      expr
                        | non_unary_expr GE       expr
                        | non_unary_expr '~'      expr
                        | non_unary_expr NO_MATCH expr
                        | non_unary_expr In NAME
                        | '(' multiple_expr_list ')' In NAME
                        | non_unary_expr AND newline_opt expr
                        | non_unary_expr OR  newline_opt expr
                        | non_unary_expr '?' expr ':' expr
                        | NUMBER
                        | STRING
                        | lvalue
                        | ERE
                        | lvalue INCR
                        | lvalue DECR
                        | INCR lvalue
                        | DECR lvalue
                        | lvalue POW_ASSIGN expr
                        | lvalue MOD_ASSIGN expr
                        | lvalue MUL_ASSIGN expr
                        | lvalue DIV_ASSIGN expr
                        | lvalue ADD_ASSIGN expr
                        | lvalue SUB_ASSIGN expr
                        | lvalue '=' expr
                        | FUNC_NAME '(' expr_list_opt ')'
                              /* no white space allowed before '(' */
                        | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
```

```
                        | BUILTIN_FUNC_NAME
                        | non_unary_input_function
                        ;
print_expr_list_opt     : /* empty */
                        | print_expr_list
                        ;
print_expr_list         : print_expr
                        | print_expr_list ',' newline_opt print_expr
                        ;
print_expr              : unary_print_expr
                        | non_unary_print_expr
                        ;
unary_print_expr        : '+' print_expr
                        | '-' print_expr
                        | unary_print_expr '^'      print_expr
                        | unary_print_expr '*'      print_expr
                        | unary_print_expr '/'      print_expr
                        | unary_print_expr '%'      print_expr
                        | unary_print_expr '+'      print_expr
                        | unary_print_expr '-'      print_expr
                        | unary_print_expr          non_unary_print_expr
                        | unary_print_expr '~'      print_expr
                        | unary_print_expr NO_MATCH print_expr
                        | unary_print_expr In NAME
                        | unary_print_expr AND newline_opt print_expr
                        | unary_print_expr OR  newline_opt print_expr
                        | unary_print_expr '?' print_expr ':' print_expr
                        ;
non_unary_print_expr    : '(' expr ')'
                        | '!' print_expr
                        | non_unary_print_expr '^'      print_expr
                        | non_unary_print_expr '*'      print_expr
                        | non_unary_print_expr '/'      print_expr
                        | non_unary_print_expr '%'      print_expr
                        | non_unary_print_expr '+'      print_expr
                        | non_unary_print_expr '-'      print_expr
                        | non_unary_print_expr          non_unary_print_expr
                        | non_unary_print_expr '~'      print_expr
                        | non_unary_print_expr NO_MATCH print_expr
                        | non_unary_print_expr In NAME
                        | '(' multiple_expr_list ')' In NAME
                        | non_unary_print_expr AND newline_opt print_expr
                        | non_unary_print_expr OR  newline_opt print_expr
                        | non_unary_print_expr '?' print_expr ':' print_expr
                        | NUMBER
                        | STRING
                        | lvalue
                        | ERE
                        | lvalue INCR
                        | lvalue DECR
                        | INCR lvalue
                        | DECR lvalue
                        | lvalue POW_ASSIGN print_expr
                        | lvalue MOD_ASSIGN print_expr
                        | lvalue MUL_ASSIGN print_expr
                        | lvalue DIV_ASSIGN print_expr
```

```
                        | lvalue ADD_ASSIGN print_expr
                        | lvalue SUB_ASSIGN print_expr
                        | lvalue '=' print_expr
                        | FUNC_NAME '(' expr_list_opt ')'
                              /* no white space allowed before '(' */
                        | BUILTIN_FUNC_NAME '(' expr_list_opt ')'
                        | BUILTIN_FUNC_NAME
                        ;
lvalue                  : NAME
                        | NAME '[' expr_list ']'
                        | '$' expr
                        ;
non_unary_input_function: simple_get
                        | simple_get '<' expr
                        | non_unary_expr '|' simple_get
                        ;
unary_input_function    : unary_expr '|' simple_get
                        ;
simple_get              : GETLINE
                        | GETLINE lvalue
                        ;
newline_opt             : /* empty */
                        | newline_opt NEWLINE
                        ;
```

This grammar has several ambiguities that are resolved as follows:

- Operator precedence and associativity are as described in Table 3-1 on page 108.
- In case of ambiguity, an **else** will be associated with the most immediately preceding **if** that would satisfy the grammar.
- In some contexts, a slash (/) that is used to surround an ERE could also be the division operator. This is resolved in such a way that wherever the division operator could appear, a slash is assumed to be the division operator. (There is no unary division operator.)

One convention that might not be obvious from the formal grammar is where newline characters are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape newline characters between any lexical tokens. In addition, newline characters without backslashes can follow a comma, an open brace, logical AND operator (**&&**), logical OR operator (**| |**), the **do** keyword, the **else** keyword, and the closing parenthesis of an **if**, **for** or **while** statement. For example:

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

Lexical Conventions

The lexical conventions for *awk* programs, with respect to the preceding grammar, are as follows:

1. Except as noted, *awk* will recognise the longest possible token or delimiter beginning at a given point.
2. A comment consists of any characters beginning with the number sign character and terminated by, but excluding the next occurrence of, a newline character. Comments will have no effect, except to delimit lexical tokens.

3. The character newline will be recognised as the token **NEWLINE**.
4. A backslash character immediately followed by a newline character will have no effect.
5. The token **STRING** represents a string constant. A string constant begins with the character ". Within a string constant, a backslash character will be considered to begin an escape sequence as specified in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v). In addition, the escape sequences in Table 3-2 on page 113 will be recognised. A newline character will not occur within a string constant. A string constant will be terminated by the first unescaped occurrence of the character " after the one that begins the string constant. The value of the string will be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting " characters.
6. The token **ERE** represents an extended regular expression constant. An ERE constant begins with the slash character. Within an ERE constant, a backslash character will be considered to begin an escape sequence as specified in the table in the **XBD** specification, **Chapter 3**, **File Format Notation**. In addition, the escape sequences in Table 3-2 on page 113 will be recognised. A newline character must not occur within an ERE constant. An ERE constant will be terminated by the first unescaped occurrence of the slash character after the one that begins the string constant. The extended regular expression represented by the ERE constant will be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting slash characters.
7. A blank character has no effect, except to delimit lexical tokens or within **STRING** or **ERE** tokens.
8. The token **NUMBER** represents a numeric constant. Its form and numeric value are equivalent to either of the tokens **floating-constant** or **integer-constant** as specified by the ISO C standard, with the following exceptions:
   a. An integer constant cannot begin with 0x or include the hexadecimal digits a, b, c, d, e, f, A, B, C, D, E or F.
   b. The value of an integer constant beginning with 0 will be taken in decimal rather than octal.
   c. An integer constant cannot include a suffix (u, U, l or L).
   d. A floating constant cannot include a suffix (f, F, l or L).

   If the value is too large or too small to be representable, the behaviour is undefined.
9. A sequence of underscores, digits and alphabetics from the portable character set (see the **XBD** specification, **Section 4.1**, **Portable Character Set**), beginning with an underscore or alphabetic, will be considered a word.
10. The following words are keywords that will be recognised as individual tokens; the name of the token is the same as the keyword:

BEGIN	**delete**	**for**	**in**	**printf**
END	**do**	**function**	**next**	**return**
break	**else**	**getline**	**print**	**while**
continue	**exit**	**if**		

11. The following words are names of built-in functions and will be recognised as the token **BUILTIN_FUNC_NAME**:

atan2	**index**	**match**	**sprintf**	**substr**
close	**int**	**rand**	**sqrt**	**system**
cos	**length**	**sin**	**srand**	**tolower**
exp	**log**	**split**	**sub**	**toupper**
gsub				

    The above-listed keywords and names of built-in functions are considered reserved words.

12. The token **NAME** consists of a word that is not a keyword or a name of a built-in function and is not followed immediately (without any delimiters) by the (character.

13. The token **FUNC_NAME** consists of a word that is not a keyword or a name of a built-in function, followed immediately (without any delimiters) by the (character. The (character will not be included as part of the token.

14. The following two-character sequences will be recognised as the named tokens:

Token Name	Sequence	Token Name	Sequence
ADD_ASSIGN	**+=**	**NO_MATCH**	**!~**
SUB_ASSIGN	**-=**	**EQ**	**==**
MUL_ASSIGN	***=**	**LE**	**<=**
DIV_ASSIGN	**/=**	**GE**	**>=**
MOD_ASSIGN	**%=**	**NE**	**!=**
POW_ASSIGN	**^=**	**INCR**	**++**
OR	**||**	**DECR**	**--**
AND	**&&**	**APPEND**	**>>**

15. The following single characters will be recognised as tokens whose names are the character:

```
<newline> { } ( ) [ ] , ; + - * % ^ ! > < | ? : ~ $ =
```

There is a lexical ambiguity between the token **ERE** and the tokens / and **DIV_ASSIGN**. When an input sequence begins with a slash character in any syntactic context where the token / or **DIV_ASSIGN** could appear as the next token in a valid program, the longer of those two tokens that can be recognised will be recognised. In any other syntactic context where the token **ERE** could appear as the next token in a valid program, the token **ERE** will be recognised.

EXIT STATUS

The following exit values are returned:

0 All input files were processed successfully.
>0 An error occurred.

The exit status can be altered within the program by using an **exit** expression.

CONSEQUENCES OF ERRORS

If any *file* operand is specified and the named file cannot be accessed, *awk* will write a diagnostic message to standard error and terminate without any further action.

If the program specified by either the *program* operand or a *progfile* operand is not a valid *awk* program (as specified in **EXTENDED DESCRIPTION**), the behaviour is undefined.

APPLICATION USAGE

The **index**, **length**, **match** and **substr** functions should not be confused with similar functions in the ISO C standard; the *awk* versions deal with characters, while the ISO C standard deals with bytes.

Because the concatenation operation is represented by adjacent expressions rather than an explicit operator, it is often necessary to use parentheses to enforce the proper evaluation precedence.

EXAMPLES

The *awk* program specified in the command line is most easily specified within single-quotes (for example, *'program'*) for applications using *sh*, because *awk* programs commonly contain characters that are special to the shell, including double-quotes. In the cases where an *awk* program contains single-quote characters, it is usually easiest to specify most of the program as strings within single-quotes concatenated by the shell with quoted single-quote characters. For example:

```
awk '/'\''/ { print "quote:", $0 }'
```

prints all lines from the standard input containing a single-quote character, prefixed with quote:.

The following are examples of simple *awk* programs:

1. Write to the standard output all input lines for which field 3 is greater than 5:

   ```
   $3 > 5
   ```

2. Write every tenth line:

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

3. Write any line with a substring matching the regular expression:

   ```
   /(G|D)(2[0-9][[:alpha:]]*)/
   ```

4. Print any line with a substring containing a G or D, followed by a sequence of digits and characters. This example uses character classes **digit** and **alpha** to match language-independent digit and alphabetic characters respectively:

   ```
   /(G|D)([[:digit:][:alpha:]]*)/
   ```

5. Write any line in which the second field matches the regular expression and the fourth field does not:

   ```
   $2 ~ /xyz/ && $4 !~ /xyz/
   ```

6. Write any line in which the second field contains a backslash:

   ```
   $2 ~ /\\/
   ```

7. Write any line in which the second field contains a backslash. Note that backslash escapes are interpreted twice, once in lexical processing of the string and once in processing the regular expression:

   ```
   $2 ~ "\\\\"
   ```

8. Write the second to the last and the last field in each line. Separate the fields by a colon:

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

9. Write the line number and number of fields in each line. The three strings representing the line number, the colon and the number of fields are concatenated and that string is written to standard output:

```
{print NR ":" NF}
```

10. Write lines longer than 72 characters:

```
length($0) > 72
```

11. Write first two fields in opposite order separated by the **OFS**:

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

12. Same, with input fields separated by comma or space and tab characters, or both:

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

13. Add up first column, print sum and average:

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

14. Write fields in reverse order, one per line (many lines out for each line in):

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

15. Write all lines between occurrences of the strings **start** and **stop**:

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

16. Write all lines whose first field is different from the previous one:

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

17. Simulate *echo:*

```
BEGIN  {
       for (i = 1; i < ARGC; ++i)
              printf("%s%s", ARGV[i], i==ARGC-1?"\n":" ")
}
```

18. Write the path prefixes contained in the *PATH* environment variable, one per line:

```
BEGIN  {
       n = split (ENVIRON["PATH"], path, ":")
       for (i = 1; i <= n; ++i)
              print path[i]
}
```

19. If there is a file named **input** containing page headers of the form:

    Page #

    and a file named **program** that contains:

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

    then the command line:

```
awk –f program n=5 input
```

    will print the file **input**, filling in page numbers starting at 5.

FUTURE DIRECTIONS

None.

SEE ALSO

grep, *lex*, *sed*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Issue 4, Version 2

The **EXAMPLES** are corrected as follows:

- In Example 10, the braces are removed.
- In Example 17, the invocation of `printf` is corrected.

NAME

banner – make large letters (**WITHDRAWN**)

SYNOPSIS

OF `banner` *`string ...`*

APPLICATION USAGE

This utility has been withdrawn because its output could not be well-specified in an internationalised environment (particularly for large, complex character sets) and it is rarely needed by portable applications.

SEE ALSO

echo.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

This page has been marked WITHDRAWN.

NAME

basename – return non-directory portion of pathname

SYNOPSIS

`basename` *`string`* **[***`suffix`***]**

DESCRIPTION

The *string* operand will be treated as a pathname, as defined in **Pathname**. The string *string* will be converted to the filename corresponding to the last pathname component in *string* and then the suffix string *suffix*, if present, will be removed. This will be done by performing actions equivalent to the following steps in order:

1. If *string* is *//*, it is implementation-dependent whether steps 2 to 5 are skipped or processed.
2. If *string* consists entirely of slash characters, *string* will be set to a single slash character. In this case, skip steps 3 to 5.
3. If there are any trailing slash characters in *string*, they will be removed.
4. If there are any slash characters remaining in *string*, the prefix of *string* up to and including the last slash character in *string* will be removed.
5. If the *suffix* operand is present, is not identical to the characters remaining in *string*, and is identical to a suffix of the characters remaining in *string*, the suffix *suffix* will be removed from *string*. Otherwise, *string* will not be modified by this step. It will not be considered an error if *suffix* is not found in *string*.

The resulting string will be written to standard output.

OPTIONS

None.

OPERANDS

The following operands are supported:

string A string.

suffix A string.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *basename*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *basename* utility will write a line to the standard output in the following format:

```
"%s\n", <resulting string>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The definition of *pathname* specifies implementation-dependent behaviour for pathnames starting with two slash characters. Therefore, applications must not arbitrarily add slashes to the beginning of a pathname unless they can ensure that there are more or less than two or are prepared to deal with the implementation-dependent consequences.

EXAMPLES
If the string *string* is a valid pathname:

```
$(basename "string")
```

produces a filename that could be used to open the file named by *string* in the directory returned by:

```
$(dirname "string")
```

If the string *string* is not a valid pathname, the same algorithm is used, but the result need not be a valid filename. The *basename* utility is not expected to make any judgements about the validity of *string* as a pathname; it just follows the specified algorithm to produce a result string.

The following shell script compiles **/usr/src/cmd/cat.c** and moves the output to a file named **cat** in the current directory when invoked with the argument /usr/src/cmd/cat or with the argument /usr/src/cmd/cat.c:

```
c89 $(dirname "$1")/$(basename "$1" .c).c
mv a.out $(basename "$1" .c)
```

FUTURE DIRECTIONS

None.

SEE ALSO

dirname, Section 2.5 on page 27.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

batch – execute commands when the system load permits

SYNOPSIS

```
batch
```

DESCRIPTION

The *batch* utility reads commands to be executed at a later time. It is the equivalent of the command:

```
at -q b -m now
```

where queue b is a special *at* queue, specifically for batch jobs. Batch jobs will be submitted to the batch queue with no time constraints and run by the system using algorithms, based on unspecified factors, that may vary with each invocation of *batch*.

EX Users are permitted to use *batch* 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 *batch*. If neither file exists, only a process with the appropriate privileges is allowed to submit a job. If only **at.deny** exists and is empty, global usage is permitted. The **at.allow** and **at.deny** files consist of one user name per line.

OPTIONS

None.

OPERANDS

None.

STDIN

The standard input must be a text file consisting of commands acceptable to the shell command language described in Chapter 2 on page 19. The standard input will only be used if no **–f** *file* option is specified.

INPUT FILES

EX The text files **/usr/lib/cron/at.allow** and **/usr/lib/cron/at.deny** contain user names, one per line, of users who are, respectively, authorised or denied access to the *at* and *batch* utilities.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *batch*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

LC_TIME
Determine the format and contents for date and time strings written by *batch*.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

SHELL Determine the name of a command interpreter to be used to invoke the at-job. If the variable is unset or null, *sh* will be used. If it is set to a value other than a name for *sh*, the implementation will do one of the following: use that shell; use *sh*; use the login shell from the user database; any of the preceding accompanied by a warning diagnostic about which was chosen.

TZ Determine the timezone. The job will be submitted for execution at the time specified by *timespec* or **–t** *time* relative to the timezone specified by the *TZ* variable. If *timespec* specifies a timezone, it will override *TZ*. If *timespec* does not specify a timezone and *TZ* is unset or null, an unspecified default timezone will be used.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When standard input is a terminal, prompts of unspecified format for each line of the user input described in **STDIN** may be written to standard output.

In the POSIX locale, the following will be written to the standard output for each job when jobs are listed in response to the **–l** option:

```
"%s\t%s\n", at_job_id, <date>
```

where <*date*> is equivalent in format to the output of:

```
date +"%a %b %e %T %Y"
```

The date and time written will be adjusted so that they appear in the timezone of the user (as determined by the *TZ* variable).

STDERR
The following will be written to standard error when a job has been successfully submitted:

```
"job %s at %s\n", at_job_id, <date>
```

where <*date*> will have the same format as is described in **STDOUT**. Neither this, nor warning messages concerning the selection of the command interpreter, are considered a diagnostic that changes the exit status.

Diagnostic messages, if any, are written to standard error.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
The job will not be scheduled.

APPLICATION USAGE

It may be useful to redirect standard output within the specified commands.

EXAMPLES

1. This sequence can be used at a terminal:

```
batch
sort < file >outfile
EOT
```

2. 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 <<!
diff file1 file2 2>&1 >outfile | mailx mygroup
!
```

FUTURE DIRECTIONS

None.

SEE ALSO

at.

CHANGE HISTORY

First released in Issue 2.

Issue 3

References to ‘‘superuser’’ have been changed to ‘‘process with appropriate privileges’’; otherwise, functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised and separated from the *at* description. Aligned with the ISO/IEC 9945-2:1993 standard.

NAME

bc – arbitrary-precision arithmetic language

SYNOPSIS

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

DESCRIPTION

The *bc* utility implements an arbitrary precision calculator. It takes input from any files given, then reads from the standard input. If the standard input and standard output to *bc* are attached to a terminal, the invocation of *bc* is considered to be *interactive*, causing behavioural constraints described in the following sections.

OPTIONS

The *bc* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–l (The letter ell.) Define the math functions and initialise **scale** to 20, instead of the default zero. See **EXTENDED DESCRIPTION**.

OPERANDS

The following operands are supported:

file A pathname of a text file containing *bc* program statements. After all cases of *file* have been read, *bc* will read the standard input.

STDIN

See **INPUT FILES**.

INPUT FILES

Input files must be text files containing a sequence of comments, statements and function definitions that will be executed as they are read.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *bc*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output of the *bc* utility is controlled by the program read, and consists of zero or more lines containing the value of all executed expressions without assignments. The radix and precision of the output are controlled by the values of the **obase** and **scale** variables. See **EXTENDED DESCRIPTION**.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

Grammar

The grammar in this section and the lexical conventions in the following section together describe the syntax for *bc* programs. The general conventions for this style of grammar are described in Section 1.8 on page 10. A valid program can be represented as the non-terminal symbol **program** in the grammar. This formal syntax takes precedence over the preceding text syntax description.

```
%token   EOF NEWLINE STRING LETTER NUMBER

%token   MUL_OP
/*       '*', '/', '%'                              */

%token   ASSIGN_OP
/*       '=', '+=', '-=', '*=', '/=', '%=', '^=' */

%token   REL_OP
/*       '==', '<=', '>=', '!=', '<', '>'         */

%token   INCR_DECR
/*       '++', '--'                                 */

%token    Define    Break    Quit    Length
/*       'define', 'break', 'quit', 'length'      */

%token    Return    For    If    While    Sqrt
/*       'return', 'for', 'if', 'while', 'sqrt'   */

%token    Scale    Ibase    Obase    Auto
/*       'scale', 'ibase', 'obase', 'auto'        */

%start   program

%%

program              : EOF
                     | input_item program
                     ;

input_item           : semicolon_list NEWLINE
                     | function
```

```
                        ;
semicolon_list          : /* empty */
                        | statement
                        | semicolon_list ';' statement
                        | semicolon_list ';'
                        ;
statement_list          : /* empty */
                        | statement
                        | statement_list NEWLINE
                        | statement_list NEWLINE statement
                        | statement_list ';'
                        | statement_list ';' statement
                        ;
statement               : expression
                        | STRING
                        | Break
                        | Quit
                        | Return
                        | Return '(' return_expression ')'
                        | For '(' expression ';'
                            relational_expression ';'
                            expression ')' statement
                        | If '(' relational_expression ')' statement
                        | While '(' relational_expression ')' statement
                        | '{' statement_list '}'
                        ;
function                : Define LETTER '(' opt_parameter_list ')'
                            '{' NEWLINE opt_auto_define_list
                            statement_list '}'
                        ;
opt_parameter_list      : /* empty */
                        | parameter_list
                        ;
parameter_list          : LETTER
                        | define_list ',' LETTER
                        ;
opt_auto_define_list    : /* empty */
                        | Auto define_list NEWLINE
                        | Auto define_list ';'
                        ;
define_list             : LETTER
                        | LETTER '[' ']'
                        | define_list ',' LETTER
                        | define_list ',' LETTER '[' ']'
```

```
                        ;
opt_argument_list       : /* empty */
                        | argument_list
                        ;
argument_list           : expression
                        | argument_list ',' expression
                        ;
relational_expression   : expression
                        | expression REL_OP expression
                        ;
return_expression       : /* empty */
                        | expression
                        ;
expression              : named_expression
                        | NUMBER
                        | '(' expression ')'
                        | LETTER '(' opt_argument_list ')'
                        | '-' expression
                        | expression '+' expression
                        | expression '-' expression
                        | expression MUL_OP expression
                        | expression '^' expression
                        | INCR_DECR named_expression
                        | named_expression INCR_DECR
                        | named_expression ASSIGN_OP expression
                        | Length '(' expression ')'
                        | Sqrt '(' expression ')'
                        | Scale '(' expression ')'
                        ;
named_expression        : LETTER
                        | LETTER '[' expression ']'
                        | Scale
                        | Ibase
                        | Obase
                        ;
```

Lexical Conventions in bc

The lexical conventions for *bc* programs, with respect to the preceding grammar, are as follows:

1. Except as noted, *bc* recognises the longest possible token or delimiter beginning at a given point.
2. A comment consists of any characters beginning with the two adjacent characters /* and terminated by the next occurrence of the two adjacent characters */. Comments have no effect except to delimit lexical tokens.
3. The character newline is recognised as the token **NEWLINE**.

4. The token **STRING** represents a string constant; it consists of any characters beginning with the double-quote character (") and terminated by another occurrence of the double-quote character. The value of the string is the sequence of all characters between, but not including, the two double-quote characters. All characters are taken literally from the input, and there is no way to specify a string containing a double-quote character. The length of the value of each string is limited to {BC_STRING_MAX} bytes.
5. A blank character has no effect except as an ordinary character if it appears within a **STRING** token, or to delimit a lexical token other than **STRING**.
6. The combination of a backslash character immediately followed by a newline character has no effect other than to delimit lexical tokens with the following exceptions:
   - It is interpreted as the character sequence \ newline in **STRING** tokens.
   - It is ignored as part of a multi-line **NUMBER** token.
7. The token **NUMBER** represents a numeric constant. It is recognised by the following grammar:

```
NUMBER  : integer
        | '.' integer
        | integer '.'
        | integer '.' integer
        ;
integer : digit
        | integer digit
        ;
digit   : 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7
        | 8 | 9 | A | B | C | D | E | F
        ;
```

8. The value of a **NUMBER** token is interpreted as a numeral in the base specified by the value of the internal register **ibase** (described below). Each of the **digit** characters has the value from 0 to 15 in the order listed here, and the period character represents the radix point. The behaviour is undefined if digits greater than or equal to the value of **ibase** appear in the token. However, note the exception for single-digit values being assigned to **ibase** and **obase** themselves, in **Operations in bc** on page 144.
9. The following keywords are recognised as tokens:

auto	**for**	**length**	**return**	**sqrt**
break	**ibase**	**obase**	**scale**	**while**
define	**if**	**quit**		

10. Any of the following characters occurring anywhere except within a keyword are recognised as the token **LETTER**:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
```

11. The following single-character and two-character sequences are recognised as the token **ASSIGN_OP**:

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

12. If an = character, as the beginning of a token, is followed by a – character with no intervening delimiter, the behaviour is undefined.

13. The following single-characters are recognised as the token **MUL_OP**:

    ```
    *       /       %
    ```

14. The following single-character and two-character sequences are recognised as the token **REL_OP**:

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

15. The following two-character sequences are recognised as the token **INCR_DECR**:

    ```
    ++      - -
    ```

16. The following single characters are recognised as tokens whose names are the character:

    ```
    <newline>  (  )  ,  +  -  ;  [  ]  ^  {  }
    ```

17. The token **EOF** will be returned when the end of input is reached.

Operations in bc

There are three kinds of identifiers: ordinary identifiers, array identifiers and function identifiers. All three types consist of single lower-case letters. Array identifiers are followed by square brackets ([ ]). An array subscript is required except in an argument or auto list. Arrays are singly dimensioned and can contain up to {BC_DIM_MAX} elements. Indexing begins at zero so an array is indexed from 0 to {BC_DIM_MAX}–1. Subscripts will be truncated to integers. Function identifiers must be followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict.

The following table summarises the rules for precedence and associativity of all operators. Operators on the same line have the same precedence; rows are in order of decreasing precedence.

Operator	Associativity
++, ––	not applicable
unary –	not applicable
^	right to left
*, /, %	left to right
+, binary –	left to right
=, +=, –=, *=, /=, %=, ^=	right to left
==, <=, >=, !=, <, >	none

Table 3-3 Operators in *bc*

Each expression or named expression has a *scale*, which is the number of decimal digits that are maintained as the fractional portion of the expression.

Named expressions are places where values are stored. Named expressions are valid on the left side of an assignment. The value of a named expression is the value stored in the place named. Simple identifiers and array elements are named expressions; they have an initial value of zero and an initial scale of zero.

The internal registers **scale**, **ibase** and **obase** are all named expressions. The scale of an expression consisting of the name of one of these registers is zero; values assigned to any of these registers will be truncated to integers. The **scale** register contains a global value used in computing the scale of expressions (as described below). The value of the register **scale** is limited to 0 ≤ **scale** ≤ {BC_SCALE_MAX} and has a default value of zero. The **ibase** and **obase** registers are the input and output number radix, respectively. The value of **ibase** is limited to:

```
2 ≤ ibase ≤ 16
```

The value of **obase** is limited to:

```
2 ≤ obase ≤ {BC_BASE_MAX}
```

When either **ibase** or **obase** is assigned a single **digit** value from the list in **Lexical Conventions in bc** on page 142, the value is assumed in hexadecimal. (For example, ibase=A sets to base ten, regardless of the current **ibase** value.) Otherwise, the behaviour is undefined when digits greater than or equal to the value of **ibase** appear in the input. Both **ibase** and **obase** have initial values of 10.

Internal computations will be conducted as if in decimal, regardless of the input and output bases, to the specified number of decimal digits. When an exact result is not achieved, (for example, scale=0; 3.2 / 1) the result will be truncated.

For all values of **obase** specified by this document, numerical values will be output as follows:

1. If the value is less than zero, a hyphen (–) character will be output.
2. One of the following will be output, depending on the numerical value:
   - If the absolute value of the numerical value is greater than or equal to one, the integer portion of the value will be output as a series of digits appropriate to **obase** (as described below). The most significant non-zero digit will be output next, followed by each successively less significant digit.
   - If the absolute value of the numerical value is less than one but greater than zero and the scale of the numerical value is greater than zero, it is unspecified whether the character 0 is output.
   - If the numerical value is zero, the character 0 will be output.
3. If the scale of the value is greater than zero, a period character will be output, followed by a series of digits appropriate to **obase** (as described below) representing the most significant portion of the fractional part of the value. If *s* represents the scale of the value being output, the number of digits output will be *s* if **obase** is 10, less than or equal to *s* if **obase** is greater than 10, or greater than or equal to *s* if **obase** is less than 10. For **obase** values other than 10, this should be the number of digits needed to represent a precision of 10^s.

For **obase** values from 2 to 16, valid digits are the first **obase** of the single characters:

```
0   1   2   3   4   5   6   7   8   9   A   B   C   D   E   F
```

which represent the values zero to 15, inclusive, respectively.

For bases greater than 16, each digit is written as a separate multi-digit decimal number. Each digit except the most significant fractional digit will be preceded a single space character. For bases from 17 to 100, *bc* will write two-digit decimal numbers; for bases from 101 to 1000, three-digit decimal strings and so on. For example, the decimal number 1024 in base 25 would be written as:

```
␣01␣15␣24
```

in base 125, as:

```
␣008␣024
```

Very large numbers will be split across lines with 70 characters per line in the POSIX locale; other locales may split at different character boundaries. Lines that are continued must end with a backslash (\).

A function call consists of a function name followed by parentheses containing a comma-separated list of expressions, which are the function arguments. A whole array passed as an argument is specified by the array name followed by empty square brackets. All function arguments are passed by value. As a result, changes made to the formal parameters have no effect on the actual arguments. If the function terminates by executing a **return** statement, the value of the function will be the value of the expression in the parentheses of the **return** statement or will be zero if no expression is provided or if there is no **return** statement.

The result of **sqrt**(*expression*) will be the square root of the expression. The result will be truncated in the least significant decimal place. The scale of the result will be the scale of the expression or the value of **scale**, whichever is larger.

The result of **length**(*expression*) will be the total number of significant decimal digits in the expression. The scale of the result will be zero.

The result of **scale**(*expression*) will be the scale of the expression. The scale of the result will be zero.

A numeric constant will be an expression. The scale will be the number of digits that follow the radix point in the input representing the constant, or zero if no radix point appears.

The sequence **(** *expression* **)** will be an expression with the same value and scale as *expression*. The parentheses can be used to alter the normal precedence.

The semantics of the unary and binary operators are as follows:

–expression
: The result will be the negative of the *expression*. The scale of the result will be the scale of *expression*.

The unary increment and decrement operators will not modify the scale of the named expression upon which they operate. The scale of the result will be the scale of that named expression.

++named-expression
: The named expression will be incremented by one. The result will be the value of the named expression after incrementing.

––named-expression
: The named expression will be decremented by one. The result will be the value of the named expression after decrementing.

named-expression++
: The named expression will be incremented by one. The result will be the value of the named expression before incrementing.

named-expression– –
The named expression will be decremented by one. The result will be the value of the named expression before decrementing.

The exponentiation operator, circumflex (^), binds right to left.

expression^*expression*
The result will be the first *expression* raised to the power of the second *expression*. If the second expression is not an integer, the behaviour is undefined. If a is the scale of the left expression and b is the absolute value of the right expression, the scale of the result will be:

```
if b >= 0 min(a * b, max(scale, a))
if b <  0 scale
```

The multiplicative operators (*, /, %) bind left to right.

expression * *expression*
The result will be the product of the two expressions. If a and b are the scales of the two expressions, then the scale of the result will be:

```
min(a+b,max(scale,a,b))
```

expression / *expression*
The result will be the quotient of the two expressions. The scale of the result will be the value of **scale**.

expression % *expression*
For expressions *a* and *b*, a % b will be evaluated equivalent to the steps:

1. Compute a / b to current scale.
2. Use the result to compute:

   ```
   a - (a / b) * b
   ```

   to scale:

   ```
   max(scale + scale(b), scale(a))
   ```

The scale of the result will be:

```
max(scale + scale(b), scale(a))
```

When **scale** is zero, the % operator is the mathematical remainder operator.

The additive operators (+, –) bind left to right.

expression + *expression*
The result will be the sum of the two expressions. The scale of the result will be the maximum of the scales of the expressions.

expression – *expression*
The result will be the difference of the two expressions. The scale of the result will be the maximum of the scales of the expressions.

The assignment operators (=, +=, –=, *=, /=, %=, ^=) bind right to left.

named-expression = *expression*
This expression results in assigning the value of the expression on the right to the named expression on the left. The scale of both the named expression and the result will be the scale of *expression*.

The compound assignment forms:

```
named-expression <operator>= expression
```

are equivalent to:

```
named-expression = named-expression <operator> expression
```

except that the *named-expression* will be evaluated only once.

Unlike all other operators, the relational operators (<, >, <=, >=, ==, !=) will be only valid as the object of an **if**, **while** or inside a **for** statement.

expression1 < *expression2*
: The relation will be true if the value of *expression1* is strictly less than the value of *expression2*.

expression1 > *expression2*
: The relation will be true if the value of *expression1* is strictly greater than the value of *expression2*.

expression1 <= *expression2*
: The relation will be true if the value of *expression1* is less than or equal to the value of *expression2*.

expression1 >= *expression2*
: The relation will be true if the value of *expression1* is greater than or equal to the value of *expression2*.

expression1 == *expression2*
: The relation will be true if the values of *expression1* and *expression2* are equal.

expression1 != *expression2*
: The relation will be true if the values of *expression1* and *expression2* are unequal.

There are only two storage classes in *bc*, global and automatic (local). Only identifiers that are to be local to a function need be declared with the **auto** command. The arguments to a function will be local to the function. All other identifiers are assumed to be global and available to all functions. All identifiers, global and local, have initial values of zero. Identifiers declared as auto will be allocated on entry to the function and released on returning from the function. They therefore do not retain values between function calls. Auto arrays will be specified by the array name followed by empty square brackets. On entry to a function, the old values of the names that appear as parameters and as automatic variables are pushed onto a stack. Until the function returns, reference to these names refers only to the new values.

References to any of these names from other functions that are called from this function also refer to the new value until one of those functions uses the same name for a local variable.

When a statement is an expression, unless the main operator is an assignment, execution of the statement will write the value of the expression followed by a newline character.

When a statement is a string, execution of the statement will write the value of the string.

Statements separated by semicolons or newline characters will be executed sequentially. In an interactive invocation of *bc*, each time a newline character is read that satisfies the grammatical production:

```
input_item : semicolon_list NEWLINE
```

the sequential list of statements making up the **semicolon_list** will be executed immediately and any output produced by that execution will be written without any delay due to buffering.

In an **if** statement (**if** (*relation*) *statement*), the *statement* will be executed if the relation is true.

The **while** statement (**while** (*relation*) *statement*) implements a loop in which the *relation* is tested; each time the *relation* is true, the *statement* will be executed and the *relation* retested. When the *relation* is false, execution will resume after *statement*.

A **for** statement (**for** (*expression*; *relation*; *expression*) *statement*) is the same as:

```
first-expression
while (relation) {
        statement
        last-expression
}
```

All three expressions must be present.

The **break** statement causes termination of a **for** or **while** statement.

The **auto** statement (**auto** *identifier***[**,*identifier***]** ...) will cause the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers are specified by following the array name by empty square brackets. The **auto** statement must be the first statement in a function definition.

A **define** statement:

```
define LETTER ( opt_parameter_list ) {
        opt_auto_define_list
        statement_list
}
```

defines a function named *LETTER*. If a function named *LETTER* was previously defined, the **define** statement will replace the previous definition. The expression:

```
LETTER ( opt_argument_list )
```

will invoke the function named *LETTER*. The behaviour is undefined if the number of arguments in the invocation does not match the number of parameters in the definition. Functions will be defined before they are invoked. A function will be considered to be defined within its own body, so recursive calls are valid. The values of numeric constants within a function will be interpreted in the base specified by the value of the **ibase** register when the function is invoked.

The **return** statements (**return** and **return**(*expression*)) will cause termination of a function, popping of its auto variables and specifies the result of the function. The first form is equivalent to return(0). The value and scale of an invocation of the function will be the value and scale of the expression in parentheses.

The **quit** statement (**quit**) will stop execution of a *bc* program at the point where the statement occurs in the input, even if it occurs in a function definition, or in an **if**, **for** or **while** statement.

The following functions will be defined when the **–l** option is specified:

s (*expression*)
: Sine of argument in radians.

c (*expression*)
: Cosine of argument in radians.

a (*expression*)
: Arctangent of argument.

l (*expression*)
Natural logarithm of argument.

e (*expression*)
Exponential function of argument.

j (*expression* , *expression*)
Bessel function of integer order.

The scale of an invocation of each of these functions will be the value of the **scale** register when the function is invoked. The behaviour is undefined if any of these functions is invoked with an argument outside the domain of the mathematical function.

EXIT STATUS

The following exit values are returned:

0	All input files were processed successfully.
unspecified	An error occurred.

CONSEQUENCES OF ERRORS

If any *file* operand is specified and the named file cannot be accessed, *bc* will write a diagnostic message to standard error and terminate without any further action.

In an interactive invocation of *bc*, the utility should print an error message and recover following any error in the input. In a non-interactive invocation of *bc*, invalid input causes undefined behaviour.

APPLICATION USAGE

Automatic variables in *bc* do not work in exactly the same way as in either C or PL/1.

For historical reasons, the exit status from *bc* cannot be relied upon to indicate that an error has occurred. Returning zero after an error is possible. Therefore, *bc* should be used primarily by interactive users (who can react to error messages) or by application programs that can somehow validate the answers returned as not including error messages.

The *bc* utility always uses the period (.) character to represent a radix point, regardless of any decimal-point character specified as part of the current locale. In languages like C or *awk*, the period character is used in program source, so it can be portable and unambiguous, while the locale-specific character is used in input and output. Because there is no distinction between source and input in *bc*, this arrangement would not be possible. Using the locale-specific character in *bc*'s input would introduce ambiguities into the language; consider the following example in a locale with a comma as the decimal-point character:

```
define f(a,b) {
       ...
}
...
f(1,2,3)
```

Because of such ambiguities, the period character is used in input. Having input follow different conventions from output would be confusing in either pipeline usage or interactive usage, so the period is also used in output.

EXAMPLES

In the shell, the following assigns an approximation of the first ten digits of π to the variable *x*:

```
x=$(printf "%s\n" 'scale = 10; 104348/33215' | bc)
```

The following *bc* program prints the same approximation of π, with a label, to standard output:

```
scale = 10
"pi equals "
104348 / 33215
```

The following defines a function to compute an approximate value of the exponential function (note that such a function is predefined if the **–l** option is specified):

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

The following prints approximate values of the exponential function of the first ten integers:

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

FUTURE DIRECTIONS

None.

SEE ALSO

awk.

CHANGE HISTORY

First released in Issue 4.

NAME

bg – run jobs in the background

SYNOPSIS

JC `bg [job_id ...]`

DESCRIPTION

If job control is enabled (see the description of *set* –**m**), the *bg* utility resumes suspended jobs from the current environment (see Section 2.12 on page 63) by running them as background jobs. If the job specified by *job_id* is already a running background job, the *bg* utility has no effect and will exit successfully.

Using *bg* to place a job into the background causes its process ID to become ‘‘known in the current shell execution environment’’, as if it had been started as an asynchronous list; see Section 2.9.3 on page 50.

OPTIONS

None.

OPERANDS

The following operand is supported:

job_id Specify the job to be resumed as a background job. If no *job_id* operand is given, the most recently suspended job is used. The format of *job_id* is described in the entry for **job control job ID** in the **XBD** specification, **Chapter 2**, **Glossary**.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *bg*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output of *bg* consists of a line in the format:

```
"[%d] %s\n", <job-number>, <command>
```

where the fields are as follows:

<*job-number*>
: A number that can be used to identify the job to the *wait*, *fg* and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with %.

<*command*>
: The associated command that was given to the shell.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If job control is disabled, the *bg* utility will exit with an error and no job will be placed in the background.

APPLICATION USAGE

A job is generally suspended by typing the SUSP character (<control>-Z on most systems); see the **XBD** specification, **Chapter 9**, **General Terminal Interface**. At that point, *bg* can put the job into the background. This is most effective when the job is expecting no terminal input and its output has been redirected to non-terminal files. A background job can be forced to stop when it has terminal output by issuing the command:

```
stty tostop
```

A background job can be stopped with the command:

```
kill -s stop job ID
```

The *bg* utility will not work as expected when it is operating in its own utility execution environment because that environment will have no suspended jobs. In the following examples:

```
... | xargs bg
(bg)
```

each *bg* operates in a different environment and will not share its parent shell's understanding of jobs. For this reason, *bg* is generally implemented as a shell regular built-in.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fg, *kill*, *jobs*, *wait*.

CHANGE HISTORY

First released in Issue 4.

NAME

c89 – compile standard C programs

SYNOPSIS

```
c89 [-c][-D name[=value]]...[-E][-g][-I directory] ... [-L directory]
... [-o outfile][-O][-s][-U name]... operand ...
```

DESCRIPTION

The *c89* utility is an interface to the standard C compilation system; it will accept source code conforming to the ISO C standard. The system conceptually consists of a compiler and link editor. The files referenced by *operand*s will be compiled and linked to produce an executable file. (It is unspecified whether the linking occurs entirely within the operation of *c89*; some systems may produce objects that are not fully resolved until the file is executed.)

If the **–c** option is specified, for all pathname operands of the form *file*.c, the files:

```
$(basename pathname .c).o
```

will be created as the result of successful compilation. If the **–c** option is not specified, it is unspecified whether such .o files are created or deleted for the *file*.c operands.

If there are no options that prevent link editing (such as **–c** or **–E**), and all operands compile and link without error, the resulting executable file will be written according to the **–o** *outfile* option (if present) or to the file **a.out**.

The executable file will be created as specified in the **XSH** specification, except that the file permissions will be set to:

S_IRWXO | S_IRWXG | S_IRWXU

and that the bits specified by the *umask* of the process will be cleared.

OPTIONS

The *c89* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that:

- The **–l** *library* operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.
- The order of specifying the **–I** and **–L** options is significant.
- Portable applications must specify each option separately; that is, grouping option letters (for example, **–cO**) need not be recognised by all implementations.

The following options are supported:

–c Suppress the link-edit phase of the compilation, and do not remove any object files that are produced.

–g Produce symbolic information in the object or executable files; the nature of this information is unspecified, and may be modified by implementation-dependent interactions with other options.

–s Produce object or executable files, or both, from which symbolic and other information not required for proper execution using the **XSH** specification *exec* family has been removed (stripped). If both **–g** and **–s** options are present, the action taken is unspecified.

–**o** *outfile*
Use the pathname *outfile*, instead of the default **a.out**, for the executable file produced. If the –**o** option is present with –**c** or –**E**, the result is unspecified.

–**D** *name*[=*value*]
Define *name* as if by a C-language **#define** directive. If no =*value* is given, a value of 1 will be used. The –**D** option has lower precedence than the –**U** option. That is, if *name* is used in both a –**U** and a –**D** option, *name* will be undefined regardless of the order of the options. Additional implementation-dependent *names* may be provided by the compiler. Implementations support at least 2048 bytes of –**D** definitions and 256 *names*.

–**E** Copy C-language source files to standard output, expanding all preprocessor directives; no compilation will be performed. If any operand is not a text file, the effects are unspecified.

–**I** *directory*
Change the algorithm for searching for headers whose names are not absolute pathnames to look in the directory named by the *directory* pathname before looking in the usual places. Thus, headers whose names are enclosed in double-quotes ("") 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 the usual places. For headers whose names are enclosed in angle brackets (< >), the header will be searched for only in directories named in –**I** options and then in the usual places. Directories named in –**I** options will be searched in the order specified. Implementations support at least ten instances of this option in a single *c89* command invocation.

–**L** *directory*
Change the algorithm of searching for the libraries named in the –**l** objects to look in the directory named by the *directory* pathname before looking in the usual places. Directories named in –**L** options will be searched in the order specified. Implementations support at least ten instances of this option in a single *c89* command invocation. If a directory specified by a –**L** option contains files named **libc.a**, **libm.a**, **libl.a** or **liby.a**, the results are unspecified.

–**O** Optimise. The nature of the optimisation is unspecified.

–**U** *name*
Remove any initial definition of *name*.

Multiple instances of the –**D**, –**I**, –**U** and –**L** options can be specified.

OPERANDS

An *operand* is either in the form of a pathname or the form –**l** *library*. At least one operand of the pathname form must be specified. The following operands are supported:

file.**c** A C-language source file to be compiled and optionally linked. The operand must be of this form if the –**c** option is used.

file.**a** A library of object files typically produced by the *ar* utility, and passed directly to the link editor. Implementations may recognise implementation-dependent suffixes other than .a as denoting object file libraries.

file.**o** An object file produced by *c89* –**c** and passed directly to the link editor. Implementations may recognise implementation-dependent suffixes other than .o as denoting object files.

The processing of other files is implementation-dependent.

–l *library*

(The letter ell.) Search the library named:

`lib`*`library`*`.a`

A library will be searched when its name is encountered, so the placement of a **–l** operand is significant. Several standard libraries can be specified in this manner, as described in **EXTENDED DESCRIPTION**. Implementations may recognise implementation-dependent suffixes other than .a as denoting libraries.

STDIN

Not used.

INPUT FILES

The input file must be one of the following: a text file containing a C-language source program; an object file in the format produced by *c89* **–c** or a library of object files, in the format produced by archiving zero or more object files, using *ar*. Implementations may supply additional utilities that produce files in these formats. Additional input file formats are implementation-dependent.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *c89*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TMPDIR

EX Provide a pathname that will override the default directory for temporary files, if any.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If more than one file operand ending in .c (or possibly other unspecified suffixes) is given, for each such file:

`"%s:\n", <`*`file`*`>`

may be written. These messages, if written, will precede the processing of each input file; they will not be written to the standard output if they are written to the standard error, as described in **STDERR**.

If the **–E** option is specified, the standard output will be a text file that represents the results of the preprocessing stage of the language; it may contain extra information appropriate for subsequent compilation passes.

STDERR

Used only for diagnostic messages. If more than one file operand ending in .c (or possibly other unspecified suffixes) is given, for each such file:

```
"%s:\n", <file>
```

may be written to allow identification of the diagnostic and warning messages with the appropriate input file. These messages, if written, will precede the processing of each input file; they will not be written to the standard error if they are written to the standard output, as described in **STDOUT**.

This utility may produce warning messages about certain conditions that do not warrant returning an error (non-zero) exit value.

OUTPUT FILES

Object files or executable files or both are produced in unspecified formats.

EXTENDED DESCRIPTION

Standard Libraries

The *c89* utility recognises the following **–l** operands for standard libraries:

–l c This operand makes visible all library functions referenced in the **XSH** specification (except for those labelled X/OPEN UNIX and except for portions marked with the UX margin legend), and except for those functions listed as residing in **<math.h>**. This operand is not required to be present to cause a search of this library.

UX If the implementation defines _XOPEN_UNIX and if the application defines the _XOPEN_SOURCE_EXTENDED feature test macro, then **–l c** also makes visible all library functions referenced in the **XSH** specification and labelled X/OPEN UNIX, and portions marked with the UX margin legend, except for those functions listed as residing in **<math.h>**.

–l m This operand makes visible all functions referenced in **<math.h>**. An implementation may search this library in the absence of this operand.

–l l This operand makes visible all functions required by the C-language output of *lex* that are not made available through the **–l c** operand.

–l y This operand makes visible all functions required by the C-language output of *yacc* that are not made available through the **–l c** operand.

In the absence of options that inhibit invocation of the link editor, such as **–c** or **–E**, the *c89* utility will cause the equivalent of a **–l c** operand to be passed to the link editor as the last **–l** operand, causing it to be searched after all other object files and libraries are loaded.

It is unspecified whether the libraries **libc.a**, **libm.a**, **libl.a** or **liby.a** exist as regular files. The implementation may accept as **–l** operands names of objects that do not exist as regular files.

UX An application that uses any API specified as X/OPEN UNIX or relies on any portion of an X/Open specification marked with the UX margin legend must define _XOPEN_SOURCE_EXTENDED = 1 in each source file or as part of its compilation environment. When _XOPEN_SOURCE_EXTENDED = 1 is defined in a source file, it must appear before any header is included.

External Symbols

The C compiler and link editor support the significance of external symbols up to a length of at least 31 bytes; the action taken upon encountering symbols exceeding the implementation-dependent maximum symbol length is unspecified.

The compiler and link editor support a minimum of 511 external symbols per source or object file, and a minimum of 4095 external symbols in total. A diagnostic message will be written to the standard output if the implementation-dependent limit is exceeded; other actions are unspecified.

EXIT STATUS

The following exit values are returned:

0 Successful compilation or link edit.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When *c89* encounters a compilation error that causes an object file not to be created, it will write a diagnostic to standard error and continue to compile other source code operands, but it will not perform the link phase and will return a non-zero exit status. If the link edit is unsuccessful, a diagnostic message will be written to standard error and *c89* will exit with a non-zero status. A portable application must rely on the exit status of *c89*, rather than on the existence or mode of the executable file.

APPLICATION USAGE

Since the *c89* utility usually creates files in the current directory during the compilation process, it is typically necessary to run the *c89* utility in a directory in which a file can be created.

On systems conforming to the ISO/IEC 9945-2:1993 standard, *c89* may be provided only as part of the C-Language Development Option; X/Open systems always provide *c89*.

Some historical implementations have created **.o** files when **–c** is not specified and more than one source file is given. Since this area is left unspecified, the application cannot rely on **.o** files being created, but it also must be prepared for any related **.o** files that already exist being deleted at the completion of the link edit.

Some historical implementations have permitted **–L** options to be interspersed with **–l** operands on the command line. For an application to compile consistently on systems that do not behave like this, it is necessary for a portable application to supply all **–L** options before any of the **–l** options.

There is the possible implication that if a user supplies versions of the standard library functions (before they would be encountered by an implicit **–l c** or explicit **–l m**), that those versions would be used in place of the standard versions. There are various reasons this might not be true (functions defined as macros, manipulations for clean name space, and so forth), so the existence of files named in the same manner as the standard libraries within the **–L** directories is explicitly stated to produce unspecified behaviour.

An application strictly portable to the ISO/IEC 9945-2:1993 standard cannot rely on *TMPDIR* overriding the default temporary directory. On XSI-conformant systems, however, this will always be the case.

EXAMPLES

The following are examples of usage:

```
c89 –o foo foo.c
```

Compiles **foo.c** and creates the executable file **foo**.

`c89 –c foo.c`
Compiles **foo.c** and creates the object file **foo.o**.

`c89 foo.c`
Compiles **foo.c** and creates the executable file **a.out**.

`c89 foo.c bar.o`
Compiles **foo.c**, links it with **bar.o**, and creates the executable file **a.out**. Also creates and leaves **foo.o**.

The following examples clarify the use and interactions of **–L** options and **–l** operands:

1. Consider the case in which module **a.c** calls function *f*( ) in library **libQ.a**, and module **b.c** calls function *g*( ) in library **libp.a**. Assume that both libraries reside in **/a/b/c**. The command line to compile and link in the desired way is:

   ```
   c89 -L /a/b/c main.o a.c -l Q b.c -l p
   ```

   In this case the **–l Q** operand need only precede the first **–l p** operand, since both **libQ.a** and libp.a reside in the same directory.

2. Multiple **–L** operands can be used when library name collisions occur. Building on the previous example, suppose that the user wants to use a new **libp.a**, in **/a/a/a**, but still wants *f*( ) from **/a/b/c/libQ.a**:

   ```
   c89 -L /a/a/a -L /a/b/c main.o a.c -l Q b.c -l p
   ```

   In this example, the linker searches the **–L** options in the order specified, and finds **/a/a/a/libp.a** before **/a/b/c/libp.a** when resolving references for **b.c**. The order of the **–l** operands is still important, however.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *cc*, *nm*, *strip*, *umask*.

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

In the **Standard Libraries** subsection, the **–l c** operand describes access to traditional interfaces if _XOPEN_UNIX is defined.

NAME

cal – print calendar

SYNOPSIS

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

DESCRIPTION

The *cal* utility writes a Gregorian calendar to standard output. If the *year* operand is specified, a calendar for that year is written. If no operands are specified, a calendar for the current month is written.

OPTIONS

None.

OPERANDS

The following operands are supported:

month Specify the month to be displayed, represented as a decimal integer from 1 (January) to 12 (December). The default is the current month.

year Specify the year for which the calendar is displayed, represented as a decimal integer from 1 to 9999. The default is the current year.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cal*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME
Determine the format and contents of the calendar.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used to calculate the value of the current month.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is used to display the calendar, in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Note that:

```
cal 83
```

refers to A.D. 83, not 1983.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support mandated.

NAME

calendar – reminder service (**TO BE WITHDRAWN**)

SYNOPSIS

EX `calendar`

DESCRIPTION

The *calendar* utility consults the file **calendar** in the current directory and writes lines that contain today's or tomorrow's date anywhere in the line to standard output. On Fridays and weekends, *tomorrow* extends to the following Monday, inclusive.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

The **calendar** file in the current directory is a text file. Each line can contain text that includes a string, in any location, that is interpreted as *today*'s or *tomorrow*'s date. Month-day date formats such as *Aug. 24*, *august 24* and *8/24* are recognised.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *calendar*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME

Determine the format of the date strings recognised by the *calendar* utility.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used to qualify the date strings recognised by the *calendar* utility.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output contains all of the selected lines from the **calendar** file.

STDERR

Not used.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Some implementations support extensions that use operands. Portable applications should not use any operands to *calendar*.

EXAMPLES

None.

FUTURE DIRECTIONS

The *calendar* utility will be withdrawn from a future issue. It belongs to a class of desktop productivity tools that is outside the scope of this document and requires considerable work in making it suitable for international use.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

cancel – cancel line printer requests

SYNOPSIS

UN EX `cancel [`*`ID ...`*`] `*`printer ...`*

UN EX `cancel `*`ID ...`*`[`*`printer ...`*`]`

DESCRIPTION

The *cancel* utility cancels line printer requests that were made by an *lp* command. The cancellation of a request that is currently printing frees the printer to print its next available request.

Cancelling requests from other users requires appropriate privileges. For each request successfully cancelled by a user who did not submit the request, the submitter may be notified that the request was cancelled.

OPTIONS

None.

OPERANDS

The following operands are supported:

ID A request *ID*, as returned by *lp*. Specifying a request *ID* cancels the associated request even if it is currently printing.

printer A printer name (for a complete list of printer names, use *lpstat*). Specifying a printer cancels the request that is currently printing on that printer.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cancel*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file containing the status of each cancellation request, in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

If mail notification is used to inform users of their requests being cancelled by other users, mail files will be modified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *cancel* utility cannot reliably cancel print requests in all conceivable circumstances. When the printer is under the control of another operating system or resides on a remote system across a network, it might not be possible to affect the status of the print job after it has left the control of the local operating system. Even on local printers, spooling hardware in the printer may make it appear that the print job has been completed long before the final page is printed.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

lp, *lpstat*, *mailx*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised and separated from the *lp* description.

Internationalised environment variable support mandated.

NAME

cat – concatenate and print files

SYNOPSIS

```
cat [-u][file ...]
```

DESCRIPTION

The *cat* utility reads files in sequence and writes their contents to the standard output in the same sequence.

OPTIONS

The *cat* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–u Write bytes from the input file to the standard output without delay as each is read.

OPERANDS

The following operand is supported:

file A pathname of an input file. If no *file* operands are specified, the standard input is used. If a *file* is –, the *cat* utility will read from the standard input at that point in the sequence. The *cat* utility will not close and reopen standard input when it is referenced in this way, but will accept multiple occurrences of – as a *file* operand.

STDIN

The standard input is used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cat*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output will contain the sequence of bytes read from the input files. Nothing else will be written to the standard output.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All input files were output successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **–u** option has value in prototyping non-blocking reads from FIFOs. The intent is to support the following sequence:

```
mkfifo foo
cat –u foo > /dev/tty13 &
cat –u > foo
```

It is unspecified whether standard output is or is not buffered in the default case. This is sometimes of interest when standard output is associated with a terminal, since buffering may delay the output. The presence of the **–u** option guarantees that unbuffered I/O is available. It is implementation-dependent whether the *cat* utility buffers output if the **–u** option is not specified. Traditionally, the **–u** option is implemented using the equivalent of the **XSH** specification *setvbuf*( ) function.

EXAMPLES

The following command:

```
cat myfile
```

writes the contents of the file **myfile** to standard output.

The following command:

```
cat doc1 doc2 > doc.all
```

concatenates the files **doc1** and **doc2** and writes the result to **doc.all**.

Because of the shell language mechanism used to perform output redirection, a command such as this:

```
cat doc doc.end > doc
```

causes the original data in **doc** to be lost.

The command:

```
cat start - middle - end > file
```

when standard input is a terminal, gets two arbitrary pieces of input from the terminal with a single invocation of *cat*. Note, however, that if standard input is a regular file, this would be equivalent to the command:

```
cat start - middle /dev/null end > file
```

because the entire contents of the file would be consumed by *cat* the first time – was used as a *file* operand and an end-of-file condition would be detected immediately when – was referenced the second time.

FUTURE DIRECTIONS

None.

SEE ALSO

more.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

cc – a C-language compilation system (**TO BE WITHDRAWN**)

SYNOPSIS

EX

```
cc [-c][-C][-e epsym] [-D name[=value]]... [-E][-f][-F][-g]
[-I directory]... [-L directory]... [-o outfile][-O][-p][-P]
[-q][-r][-s][-S][-u symname]... [-U name]... [-W options]... operand...
```

DESCRIPTION

The *cc* utility is an interface to the C-language compilation system. The system conceptually consists of a preprocessor, compiler, optimiser, assembler and link editor. The *cc* utility processes the supplied options and then executes the various tools with the appropriate arguments.

The suffix of the pathname versions of an *operand* indicates how it is to be treated. See **OPERANDS**.

The files referenced by *operand*s will be compiled/assembled and linked to produce an executable file. (It is unspecified whether the linking occurs entirely within the operation of *cc*; some systems may produce objects that are not fully resolved until the file is executed.)

If the **–c** option is specified, for all pathname operands of the form *file***.c**, the files:

```
$(basename pathname .c).o
```

will be created as the result of successful compilation. Similar results occur for pathname operands of the form *file***.i** and **.s**. If the **–c** option is not specified, it is unspecified whether such **.o** files are created or deleted for these operands.

If there are no options that prevent link editing (such as **–c** or **–E**), and all operands compile and link without error, the resulting executable file will be written according to the **–o** *outfile* option (if present) or to the file **a.out**.

The executable file will be created as specified in the **XSH** specification, except that the file permissions will be set to:

S_IRWXO | S_IRWXG | S_IRWXU

and that the bits specified by the *umask* of the process will be cleared.

OPTIONS

The *cc* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that:

- The **–l** *library* operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.
- The order of specifying the **–I** and **–L** options is significant.
- Portable applications must specify each option separately; that is, grouping option letters (for example, **–cO**) need not be recognised by all implementations.

The following options are supported:

–c Suppress the link-edit phase of the compilation, and do not remove any object files that are produced.

UN **–E** Run only the preprocessor on the named C-language programs and send the result to standard output.

–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.

PI **–F** This option is reserved for implementation-specific optimisation directives.

PI OP **–g** Cause the compiler to generate additional information needed for use by a debugger (possibly *sdb*).

–o *outfile*
Use the name *outfile* instead of the default **a.out** for the executable file produced. This is a link-edit option.

–O Do compilation phase optimisation. This option will not affect **.s** files.

PI OP **–p** This option is reserved for invoking implementation-specific profiling procedures.

UN **–P** Run only the preprocessor on the named C-language programs and leave the result on corresponding files suffixed **.i**.

PI **–q** This option is reserved for specifying implementation-specific profiling directives.

UN **–S** Compile and do not assemble the named C-language programs, and leave the assembler-language output on corresponding files suffixed .s.

PI **–W** *c*,*arg*[,*arg* ...]
Pass the arguments *arg* to phase *c* where *c* is one of [p02al] indicating preprocessing, compiling, optimising, assembling or link editing phases, respectively. For example, **–Wa**,**–m** passes **–m** to the assembler phase.

The *cc* utility also recognises a number of options that it will pass (with their associated arguments) directly to another phase of the *cc* utility. The use of the **–W** option is not required for these options.

The following options are passed by *cc* (with their associated arguments) to the preprocessor phase:

–C By default, the preprocessor strips C-language style comments. If the **–C** option is specified, all comments (except those found on preprocessor directive lines) are passed along.

–D *name*[=*value*]
Define *name* as if by a C-language **#define** directive. If no =*value* is given, a value of 1 will be used. The **–D** option has lower precedence than the **–U** option. That is, if *name* is used in both a **–U** and a **–D** option, *name* will be undefined regardless of the order of the options. Additional implementation-dependent *names* may be provided by the compiler. Implementations support at least 2048 bytes of **–D** definitions and 256 *names*.

–I *directory*
Change the algorithm for searching for headers whose names are not absolute pathnames to look in the directory named by the *directory* pathname before looking in the usual places. Thus, headers whose names are enclosed in double-quotes ("") 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 the usual places. For headers whose names are enclosed in angle brackets (< >), the header will be searched for only in directories named in **–I** options and then in the usual places. Directories named in **–I** options will be searched in the order specified. Implementations support at least ten instances of this option in a single *cc* command invocation.

–U *name*

Remove any initial definition of *name*, where *name* is a reserved symbol that is predefined by the particular preprocessor.

The following options are passed by *cc* (with their associated arguments) to the link-edit phase:

–e *epsym*

Set the default entry point address for the output file to be that of the symbol *epsym*.

–L *dir*

Change the algorithm of searching for the libraries named in the **–l** objects to look in the directory named by the *directory* pathname before looking in the usual places. Directories named in **–L** options will be searched in the order specified. Implementations support at least ten instances of this option in a single *cc* command invocation. If a directory specified by a **–L** option contains files named **libc.a**, **libm.a**, **libl.a** or **liby.a**, the results are unspecified. This option is only effective if it precedes the **–l** option on the command line.

–r Retain relocation entries in the output object file. Relocation entries must be saved if the output is to become the input of a subsequent *cc* run. The link-edit phase will not complain about unresolved references and will not make the object output executable.

–s Produce object or executable files, or both, from which symbolic and other information not required for proper execution using the **XSH** specification *exec* family has been removed (stripped). If both **–g** and **–s** options are present, the action taken is unspecified.

–u *symname*

Enter *symname* as an undefined symbol into 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 loading of the first routine.

OPERANDS

An *operand* is either in the form of a pathname or the form **–l** *library*. At least one operand of the pathname form must be specified. The following operands are supported:

file.**c** A C-language source file that may be preprocessed, compiled, optimised and link edited.

file.**i** A C-language source file that has been preprocessed, and may be compiled, optimised and link edited.

file.**s** An assembly language source file that may be assembled and link edited.

file.**a** A library of object files typically produced by the *ar* utility, and passed directly to the link editor.

The operand must be one of the forms *file*.c, *file*.i or *file*.s if the **–c** option is used.

–l *library*

(The letter ell.) Search the library named:

`lib`*`library`*`.a`

A library will be searched when its name is encountered, so the placement of a **–l** operand is significant. Several standard libraries can be specified in this manner, as described in **EXTENDED DESCRIPTION**.

Other arguments are taken to be C-language compatible object programs, typically produced by an earlier *cc* run, or perhaps libraries of C-language 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-edit option is used).

The standard C-language library is automatically available to the C-language program. Other libraries may be specified explicitly using the **–l** option with *cc*.

STDIN

Not used.

INPUT FILES

The input file will be one of the following: a text file containing a C-language source program; a text file containing an (implementation-specific) assembly-language source program; an object file in the format produced by *cc* **–c** or a library of object files, in the format produced by archiving zero or more object files, using *ar*. Additional input file formats are implementation-dependent.

ENVIRONMENT VARIABLES

The following environment variable affects the execution of *cc*:

TMPDIR
: Provide a pathname that will override the default directory for temporary files, if any.

The following environment variables may affect the execution of *cc*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If more than one file operand ending in .c, .i or .s is given, for each such file:

`"%s:\n", <file>`

may be written. These messages, if written, will precede the processing of each input file; they will not be written to standard output if they are written to standard error, as described in **STDERR**.

If the **–E** option is specified, the standard output will be a text file that represents the results of the preprocessing stage of the language; it may contain extra information appropriate for subsequent compilation passes.

STDERR

Used only for diagnostic messages. If more than one file operand ending in .c (or possibly other unspecified suffixes) is given, for each such file:

```
"%s:\n", <file>
```

may be written to allow identification of the diagnostic and warning messages with the appropriate input file. These messages, if written, will precede the processing of each input file; they will not be written to standard error if they are written to standard output, as described in **STDOUT**.

This utility may produce warning messages about certain conditions that do not warrant returning an error (non-zero) exit value.

OUTPUT FILES

If the **–P** option is specified, text files are created that represent the results of the preprocessing stage of the language.

Object files or executable files or both are produced in unspecified formats.

EXTENDED DESCRIPTION

Standard Libraries

The *cc* utility recognises the following **–l** operands for standard libraries:

–l c This operand makes visible all library functions referenced in the **XSH** specification (except for those labelled X/OPEN UNIX and except for portions marked with the UX margin legend), and except for those functions listed as residing in **<math.h>**. This operand is not required to be present to cause a search of this library.

UX If the implementation defines _XOPEN_UNIX and if the application defines the _XOPEN_SOURCE_EXTENDED feature test macro, then **–l c** also makes visible all library functions referenced in the **XSH** specification and labelled X/OPEN UNIX, and portions marked with the UX margin legend, except for those functions listed as residing in **<math.h>**.

–l m This operand makes visible all functions referenced in **<math.h>**. An implementation may search this library in the absence of this operand.

–l l This library contains all functions required by the C-language output of *lex* that are not made available through the **–l c** operand.

–l y This operand makes visible all functions required by the C-language output of *yacc* that are not made available through the **–l c** operand.

In the absence of options that inhibit invocation of the link editor, such as **–c** or **–E**, the *cc* utility will cause the equivalent of a **–l c** operand to be passed to the link editor as the last **–l** operand, causing it to be searched after all other object files and libraries are loaded.

It is unspecified whether the libraries **libc.a**, **libm.a**, **libl.a** or **liby.a** exist as regular files. The implementation may accept as **–l** operands names of objects that do not exist as regular files.

UX An application that uses any API specified as X/OPEN UNIX or relies on any portion of an X/Open specification marked with the UX margin legend must define _XOPEN_SOURCE_EXTENDED = 1 in each source file or as part of its compilation

environment. When _XOPEN_SOURCE_EXTENDED = 1 is defined in a source file, it must appear before any header is included.

External Symbols

The C compiler and link editor support the significance of external symbols up to a length of at least 31 bytes; the action taken upon encountering symbols exceeding the implementation-dependent maximum symbol length is unspecified.

The compiler and link editor support a minimum of 511 external symbols per source or object file, and a minimum of 4095 external symbols total. A diagnostic message will be written to the standard output if the implementation-dependent limit is exceeded; other actions are unspecified.

EXIT STATUS

The following exit values are returned:

0 Successful compilation or link edit.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When *cc* encounters a compilation error that causes an object file not to be created, it will write a diagnostic to standard error and continue to compile other source code operands, but it will not perform the link phase and will return a non-zero exit status. If the link edit is unsuccessful, a diagnostic message will be written to standard error and *cc* will exit with a non-zero status. A portable application must rely on the exit status of *cc*, rather than on the existence or mode of the executable file.

APPLICATION USAGE

The *c89* utility provides an interface to the ISO C standard, but the *cc* utility accepts an unspecified dialect of the C language: it may be Standard C, common-usage C or some other variant. Portable C programs should be written to conform to the ISO C standard and compiled with *c89*.

Since the *cc* utility usually creates files in the current directory during the compilation process, it is typically necessary to run the *cc* utility in a directory in which a file can be created.

Some historical implementations have created .o files when **–c** is not specified and more than one source file is given. Since this area is left unspecified, the application cannot rely on .o files being created, but it also must be prepared for any related .o files that already exist being deleted at the completion of the link edit.

Some historical implementations have permitted **–L** options to be interspersed with **–l** operands on the command line. For an application to compile consistently on systems that do not behave like this, it is necessary for a portable application to supply all **–L** options before any of the **–l** options.

There is the possible implication that if a user supplies versions of the standard library functions (before they would be encountered by an implicit **–l c** or explicit **–l m**), that those versions would be used in place of the standard versions. There are various reasons this might not be true (functions defined as macros, manipulations for clean name space, and so forth), so the existence of files named in the same manner as the standard libraries within the **–L** directories is explicitly stated to produce unspecified behaviour.

EXAMPLES

The following are examples of usage:

`cc –o foo foo.c bar.s`
: Compiles **foo.c**, assembles bar.s and creates the executable file **foo**.

`cc –c foo.c`
: Compiles **foo.c** and creates the object file **foo.o**.

`cc foo.c`
: Compiles **foo.c** and creates the executable file **a.out**.

`cc foo.c bar.o`
: Compiles **foo.c**, links it with **bar.o**, and creates the executable **a.out**. Also creates and leaves **foo.o**.

The following examples clarify the use and interactions of **–L** options and **–l** operands:

1. Consider the case in which module **a.c** calls function *f*( ) in library **libQ.a**, and module **b.c** calls function *g*( ) in library **libp.a**. Assume that both libraries reside in **/a/b/c**. The command line to compile and link in the desired way is:

   ```
   cc -L /a/b/c main.o a.c -l Q b.c -l p
   ```

   In this case the **–l Q** operand need only precede the first **–l p** operand, since both **libQ.a** and **libp.a** reside in the same directory.

2. Multiple **–L** operands can be used when library name collisions occur. Building on the previous example, suppose that the user now wants to use a new **libp.a**, in **/a/a/a**, but still wants *f*( ) from **/a/b/c/libQ.a**:

   ```
   cc -L /a/a/a -L /a/b/c main.o a.c -l Q b.c -l p
   ```

   In this example, the linker searches the **–L** options in the order specified, and finds **/a/a/a/libp.a** before **/a/b/c/libp.a** when resolving references for **b.c**. The order of the **–l** operands is still important, however.

FUTURE DIRECTIONS

This utility will be withdrawn from a future issue. The *c89* utility should be used instead.

SEE ALSO

ar, *c89*, *nm*, *sdb*, *strip*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Internationalised environment variable support made optional.

Utility Syntax Guidelines support mandated.

Issue 4, Version 2

In the **Standard Libraries** subsection, the **–l c** operand describes access to traditional interfaces if _XOPEN_UNIX is defined.

NAME

cd – change working directory

SYNOPSIS

`cd [`*`directory`*`]`

EX `cd -`

DESCRIPTION

The *cd* utility will change the working directory of the current shell execution environment; see
EX Section 2.12 on page 63. If the current working directory is successfully changed, it will save an absolute pathname of the old working directory in the environment variable *OLDPWD* and it will save an absolute pathname of the new working directory in the environment variable *PWD*.

When invoked with no operands, and the *HOME* environment variable is set to a non-empty value, the directory named in the *HOME* environment variable will become the new working directory. If *HOME* is empty or is undefined, the default behaviour is implementation-dependent.

OPTIONS

None.

OPERANDS

The following operands are supported:

directory

An absolute or relative pathname of the directory that becomes the new working directory. The interpretation of a relative pathname by *cd* depends on the *CDPATH* environment variable.

EX – When a hyphen is used as the operand, this is equivalent to the command:

```
cd "$OLDPWD" && pwd
```

which changes to the previous working directory and then writes its name.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cd*:

CDPATH

A colon-separated list of pathnames that refer to directories. If the *directory* operand does not begin with a slash (/) character, and the first component is not dot or dot-dot, *cd* will search for *directory* relative to each directory named in the *CDPATH* variable, in the order listed. The new working directory will be set to the first matching directory found. An empty string in place of a directory pathname represents the current directory. If *CDPATH* is not set, it will be treated as if it were an empty string.

HOME The name of the home directory, used when no *directory* operand is specified.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

EX *OLDPWD*
A pathname of the previous working directory, used by *cd* –.

EX *PWD* A pathname of the current working directory, set by *cd* after it has changed to that directory.

ASYNCHRONOUS EVENTS
Default.

STDOUT
EX If a non-empty directory name from *CDPATH* is used, or if *cd* – is used, an absolute pathname of the new working directory will be written to the standard output as follows:

```
"%s\n", <new directory>
```

Otherwise, there will be no output.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The directory was successfully changed.
>0 An error occurred.

CONSEQUENCES OF ERRORS
The working directory remains unchanged.

APPLICATION USAGE
Since *cd* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(cd /tmp)
nohup cd
find . -exec cd {} \;
```

it will not affect the working directory of the caller's environment.

The user must have execute (search) permission in *directory* in order to change to it.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pwd, the **XSH** specification description of *chdir*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Extensions added for *cd –*, *PWD* and *OLDPWD*.

NAME

cflow – generate C-language flowgraph (**DEVELOPMENT**)

SYNOPSIS

EX `cflow [r][-d num][-D name[=def]] ... [-i incl][-I dir] ... [-U dir] ... file ...`

DESCRIPTION

The *cflow* utility analyses a collection of object files or assembler, C-language, *lex* or *yacc* source files, and attempts to build a graph, written to standard output, charting the external references.

OPTIONS

The *cflow* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the order of the –**D**, –**I** and –**U** options (which are identical to their interpretation by *c89*) is significant. The following options are supported:

–**d** *num*
Indicate the depth at which the flowgraph is cut off. The argument *num* is a decimal integer. By default this is a very large number (typically greater than 32 000). Attempts to set the cut-off depth to a non-positive integer will be ignored.

–**i** *incl* Increase the number of included symbols. The *incl* option-argument is one of the following characters:

x Include external and static data symbols. The default is to include only functions in the flowgraph.

_ (Underscore) Include names that begin with an underscore. The default is to exclude these functions (and data if –**i x** is used).

–**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.

OPERANDS

The following operand is supported:

file The pathname of a file for which a graph is to be generated. Files suffixed in **.l**, **.y**, **.c** and **.i** are processed by *lex* and *yacc* and preprocessed by the *c89* preprocessor phase (bypassed for **.i** files) as appropriate, and then run through the first pass of *lint*. Files suffixed with **.s** are assembled and information is extracted (as in **.o** files) from the symbol table.

STDIN

Not used.

INPUT FILES

The input files are object files or assembler, C-language, *lex* or *yacc* source files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cflow*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the ordering of the output when the **–r** option is used.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The flowgraph written to standard output is formatted as follows:

```
"%d %s:%s\n", <reference number>, <global>, <definition>
```

Each line of output begins with a reference (that is, line) number, followed by a suitable amount of indentation indicating the level. This is followed by the name of the global, a colon and its definition. Normally globals are only functions not defined as an external or beginning with an underscore; see **OPTIONS** for the **–i** inclusion option. For information extracted from C-language source, the definition consists of an abstract type declaration (for example, **char** *) and, delimited by angle brackets, the name of the source file and the line number where the definition was found. Definitions extracted from object files indicate the filename and location counter under which the symbol appeared (for example, *text*).

Once a definition of a name has been written, subsequent references to that name contain only the reference number of the line where the definition can be found. For undefined references, only < > is written.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Files produced by *lex* and *yacc* cause the reordering of line number declarations, and this can confuse *cflow*. To obtain proper results, the input of *yacc* or *lex* must be directed to *cflow*.

EXAMPLES

Given the following in **file.c**:

```
int i;

main( )
{
        f( );
        g( );
        f( );
}

f( )
{
        i = h( );
}
```

The command:

```
cflow –i x file.c
```

produces the output:

```
1       main: int(), <file.c 4>
2        f: int(), <file.c 11>
3              h: <>
4              i: int, <file.c 1>
5        g: <>
```

FUTURE DIRECTIONS

None.

SEE ALSO

cc, *c89*, *lex*, *lint*, *yacc*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support mandated.

NAME

chgrp – change file group ownership

SYNOPSIS

`chgrp [-R] group file ...`

DESCRIPTION

The *chgrp* utility will set the group ID of the file named by each *file* operand to the group ID specified by the *group* operand.

For each *file* operand, it will perform actions equivalent to the **XSH** specification *chown*( ) function, called with the following arguments:

- The *file* operand will be used as the *path* argument.
- The user ID of the file will be used as the *owner* argument.
- The specified group ID will be used as the *group* argument.

Unless *chgrp* is invoked by a process with appropriate privileges, the set-user-ID and set-group-ID bits of a regular file will be cleared upon successful completion; the set-user-ID and set-group-ID bits of other file types may be cleared.

OPTIONS

The *chgrp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–R Recursively change file group IDs. For each *file* operand that names a directory, *chgrp* will change the group of the directory and all files in the file hierarchy below it.

OPERANDS

The following operands are supported:

group A group name from the group database or a numeric group ID. Either specifies a group ID to be given to each file named by one of the *file* operands. If a numeric *group* operand exists in the group database as a group name, the group ID number associated with that group name is used as the group ID.

file A pathname of a file whose group ID is to be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *chgrp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The utility executed successfully and all requested changes were made.
>0 An error occurred.

CONSEQUENCES OF ERRORS
If, when invoked with the **–R** option, *chgrp* attempts but fails to change the group ID of a particular file in a specified file hierarchy, it will continue to process the remaining files in the hierarchy. If *chgrp* cannot read or search a directory within a hierarchy, it will continue to process the other parts of the hierarchy that are accessible.

APPLICATION USAGE
Only the owner of a file or the user with appropriate privileges may change the owner or group of a file.

Some systems restrict the use of *chgrp* to a user with appropriate privileges when the *group* specified is not the effective group ID or one of the supplementary group IDs of the calling process.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
chmod, *chown*, the **XSH** specification description of *chown*( ).

CHANGE HISTORY
First released in Issue 2.

Issue 3
The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4
Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

chmod – change file modes

SYNOPSIS

`chmod [-R] mode file...`

DESCRIPTION

The *chmod* utility will change any or all of the file mode bits of the file named by each *file* operand in the way specified by the *mode* operand.

It is implementation-dependent whether and how the *chmod* utility affects any alternate or additional file access control mechanism (see **file access permissions** in the **XBD** specification, **Chapter 2**, **Glossary**) being used for the specified file.

Only a process whose effective user ID matches the user ID of the file, or a process with the appropriate privileges, will be permitted to change the file mode bits of a file.

OPTIONS

The *chmod* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–R Recursively change file mode bits. For each *file* operand that names a directory, *chmod* will change the file mode bits of the directory and all files in the file hierarchy below it.

OPERANDS

The following operands are supported:

mode Represents the change to be made to the file mode bits of each file named by one of the *file* operands; see **EXTENDED DESCRIPTION**.

file A pathname of a file whose file mode bits are to be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *chmod*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

EX The *mode* operand will be either a **symbolic_mode** expression or a non-negative octal integer. The **symbolic_mode** form is described by the grammar later in this section.

Each **clause** will specify an operation to be performed on the current file mode bits of each *file*. The operations will be performed on each *file* in the order in which the **clause**s are specified.

The *who* symbols u, g and o will specify the *user*, *group* and *other* parts of the file mode bits, respectively. A *who* consisting of the symbol a will be equivalent to **ugo**.

The *perm* symbols r, w and x represent the *read*, *write* and *execute/search* portions of file mode bits, respectively. The *perm* symbol s represent the *set-user-ID-on-execution* (when **who** contains or implies u) and *set-group-ID-on-execution* (when **who** contains or implies g) bits.

The **perm** symbol X represent the execute/search portion of the file mode bits if the file is a directory or if the current (unmodified) file mode bits have at least one of the execute bits (S_IXUSR, S_IXGRP or S_IXOTH) set. It will be ignored if the file is not a directory and none of the execute bits are set in the current file mode bits.

The **permcopy** symbols u, g and o represent the current permissions associated with the user, group and other parts of the file mode bits, respectively. For the remainder of this section, **perm** refers to the non-terminals **perm** and **permcopy** in the grammar.

If multiple **actionlist**s are grouped with a single **wholist** in the grammar, each **actionlist** will be applied in the order specified with that **wholist**. The **op** symbols represent the operation performed, as follows:

+ If **perm** is not specified, the + operation will not change the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, will be set.

Otherwise, the file mode bits represented by the specified **who** and **perm** values will be set.

– If **perm** is not specified, the – operation will not change the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, will be cleared.

Otherwise, the file mode bits represented by the specified **who** and **perm** values will be cleared.

= Clear the file mode bits specified by the **who** value, or, if no **who** value is specified, all of the file mode bits specified in this document.

If **perm** is not specified, the = operation will make no further modifications to the file mode bits.

If **who** is not specified, the file mode bits represented by **perm** for the owner, group and other permissions, except for those with corresponding bits in the file mode creation mask of the invoking process, will be set.

Otherwise, the file mode bits represented by the specified **who** and **perm** values will be set.

When using the symbolic mode form on a regular file, it is implementation-dependent whether or not:

- Requests to set the set-user-ID-on-execution or set-group-ID-on-execution bit when all execute bits are currently clear and none are being set are ignored.
- Requests to clear all execute bits also clear the set-user-ID-on-execution and set-group-ID-on-execution bits.
- Requests to clear the set-user-ID-on-execution or set-group-ID-on-execution bits when all execute bits are currently clear are ignored. However, if the command *ls* **–l** *file* writes an s in the position indicating that the set-user-ID-on-execution or set-group-ID-on-execution is set, the commands *chmod* **u–s** *file* or *chmod* **g–s** *file*, respectively, will not be ignored.

When using the symbolic mode form on other file types, it is implementation-dependent whether or not requests to set or clear the set-user-ID-on-execution or set-group-ID-on-execution bits are honoured.

If the **who** symbol o is used in conjunction with the **perm** symbol s with no other **who** symbols being specified, the set-user-ID-on-execution and set-group-ID-on-execution bits will not be modified. It will not be an error to specify the **who** symbol o in conjunction with the **perm** symbol s.

EX For an octal integer *mode* operand, the file mode bits will be set absolutely.

For each bit set in the octal number, the corresponding file permission bit shown in the following table will be set; all other file permission bits will be cleared. For regular files, for each bit set in the octal number corresponding to the set-user-ID-on-execution or the set-group-ID-on-execution bits shown in the following table will be set; if these bits are not set in the octal number, they will be cleared. For other file types, it is implementation-dependent whether or not requests to set or clear the set-user-ID-on-execution or set-group-ID-on-execution bits are honoured.

Octal	Mode bit	Octal	Mode bit	Octal	Mode bit	Octal	Mode bit
4000	S_ISUID	**0400**	S_IRUSR	**0040**	S_IRGRP	**0004**	S_IROTH
2000	S_ISGID	**0200**	S_IWUSR	**0020**	S_IWGRP	**0002**	S_IWOTH
		0100	S_IXUSR	**0010**	S_IXGRP	**0001**	S_IXOTH

When bits are set in the octal number other than those listed in the table above, the behaviour is unspecified.

Grammar for chmod

The grammar and lexical conventions in this section describe the syntax for the **symbolic_mode** operand. The general conventions for this style of grammar are described in Section 1.8 on page 10. A valid **symbolic_mode** can be represented as the non-terminal symbol **symbolic_mode** in the grammar. This formal syntax takes precedence over the preceding text syntax description.

The lexical processing will be based entirely on single characters. Implementations need not allow blank characters within the single argument being processed.

```
%start          symbolic_mode
%%

symbolic_mode : section
              | symbolic_mode ',' section
              ;

section       : actionlist
              | wholist actionlist
              ;

wholist       : who
              | wholist who
              ;

who           : 'u' | 'g' | 'o' | 'a'
              ;

actionlist    : action
              | actionlist action
              ;

action        : op
              | op permlist
              | op permcopy
              ;

permcopy      : 'u' | 'g' | 'o'
              ;

op            : '+' | '-' | '='
              ;

permlist      : perm
              | perm permlist
              ;

perm          : 'r' | 'w' | 'x' | 'X' | 's'
              ;
```

EXIT STATUS

The following exit values are returned:

0 The utility executed successfully and all requested changes were made.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If, when invoked with the **–R** option, *chmod* attempts but fails to change the mode of a particular file in a specified file hierarchy, it will continue to process the remaining files in the hierarchy, affecting the final exit status. If *chmod* cannot read or search a directory within a hierarchy, it will continue to process the other parts of the hierarchy that are accessible.

APPLICATION USAGE

The references to octal modes are marked EX because, although they are obsolescent in the ISO / IEC 9945-2: 1993 standard, XSI-conformant systems have committed to maintaining them for portable applications until further notice.

Some implementations of the *chmod* utility change the mode of a directory before the files in the directory when performing a recursive (**–R** option) change; others change the directory mode after the files in the directory. If an application tries to remove read or search permission for a

file hierarchy, the removal attempt will fail if the directory is changed first; on the other hand, trying to re-enable permissions to a restricted hierarchy will fail if directories are changed last. Users should not try to make a hierarchy inaccessible to themselves.

Some implementations of *chmod* never used the process' *umask* when changing modes; systems conformant with this document do so when **who** is not specified. Note the difference between:

```
chmod a-w file
```

which removes all write permissions, and:

```
chmod -- -w file
```

which removes write permissions that would be allowed if **file** was created with the same *umask*.

Portable applications should never assume that they know how the set-user-ID and set-group-ID bits on directories will be interpreted.

EXAMPLES

Mode	Results
a+=	Equivalent to `a+,a=`; clears all file mode bits.
go+-w	Equivalent to `go+,go-w`; clears group and other write bits.
g=o-w	Equivalent to `g=o,g-w`; sets group bit to match other bits and then clears group write bit.
g-r+w	Equivalent to `g-r,g+w`; clears group read bit and sets group write bit.
=g	Sets owner bits to match group bits and sets other bits to match group bits.

FUTURE DIRECTIONS

None.

SEE ALSO

ls, *umask*, the **XSH** specification description of *chmod*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

chown – change file ownership

SYNOPSIS

```
chown [-R] owner[:group] file ...
```

DESCRIPTION

The *chown* utility will set the user ID of the file named by each *file* operand to the user ID specified by the *owner* operand.

For each *file* operand, it will perform actions equivalent to the **XSH** specification *chown*( ) function, called with the following arguments:

1. The *file* operand will be used as the *path* argument.
2. The user ID indicated by the *owner* portion of the first operand will be used as the *owner* argument.
3. If the *group* portion of the first operand is given, the group ID indicated by it will be used as the *group* argument; otherwise, the group ID of the file will be used as the *group* argument.

Unless *chown* is invoked by a process with appropriate privileges, the set-user-ID and set-group-ID bits of a regular file will be cleared upon successful completion; the set-user-ID and set-group-ID bits of other file types may be cleared.

OPTIONS

The *chown* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–R Recursively change file user IDs, and if the *group* operand is specified, group IDs. For each *file* operand that names a directory, *chown* changes the user and group ID of the directory and all files in the file hierarchy below it.

OPERANDS

The following operands are supported:

owner[:*group*]
: A user ID and optional group ID to be assigned to *file*. The *owner* portion of this operand must be a user name from the user database or a numeric user ID. Either specifies a user ID to be given to each file named by one of the *file* operands. If a numeric *owner* operand exists in the user database as a user name, the user ID number associated with that user name will be used as the user ID. Similarly, if the *group* portion of this operand is present, it must be a group name from the group database or a numeric group ID. Either specifies a group ID to be given to each file. If a numeric group operand exists in the group database as a group name, the group ID number associated with that group name will be used as the group ID.

file A pathname of a file whose user ID is to be modified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *chown*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The utility executed successfully and all requested changes were made.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If, when invoked with the **–R** option, *chown* attempts but fails to change the user ID or, if the *group* operand is specified, group ID, of a particular file in a specified file hierarchy, it will continue to process the remaining files in the hierarchy.

If *chown* cannot read or search a directory within a hierarchy, it will continue to process the other parts of the hierarchy that are accessible.

APPLICATION USAGE

Only the owner of a file or the user with appropriate privileges may change the owner or group of a file.

Some systems restrict the use of *chown* to a user with appropriate privileges.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *chgrp*, the **XSH** specification description of *chown*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

chroot – change root directory for a command (**WITHDRAWN**)

SYNOPSIS

EX `chroot newroot command`

APPLICATION USAGE

This utility has been withdrawn because there is no portable way to set up an environment where it is useful and it is usually usable only by applications with appropriate privileges.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

This page has been marked as withdrawn. The intermediate step of publishing an issue with a **TO BE WITHDRAWN** marking has been bypassed intentionally because of the non-portable aspects of this utility.

NAME

cksum – write file checksums and sizes

SYNOPSIS

`cksum [`*`file ...`*`]`

DESCRIPTION

The *cksum* utility calculates and writes to standard output a cyclic redundancy check (CRC) for each input file, and also writes to standard output the number of octets in each file. The CRC used is based on the polynomial used for CRC error checking in the referenced Ethernet standard.

The encoding for the CRC checksum is defined by the generating polynomial:

$$G(x) = x^{32} + x^{26} + x^{23} + x^{22} + x^{16} + x^{12} + x^{11} + x^{10} + x^8 + x^7 + x^5 + x^4 + x^2 + x + 1$$

Mathematically, the CRC value corresponding to a given file is defined by the following procedure:

1. The n bits to be evaluated are considered to be the coefficients of a mod 2 polynomial $M(x)$ of degree $n–1$. These n bits are the bits from the file, with the most significant bit being the most significant bit of the first octet of the file and the last bit being the least significant bit of the last octet, padded with zero bits (if necessary) to achieve an integral number of octets, followed by one or more octets representing the length of the file as a binary value, least significant octet first. The smallest number of octets capable of representing this integer is used.
2. $M(x)$ is multiplied by x^{32} (that is, shifted left 32 bits) and divided by $G(x)$ using mod 2 division, producing a remainder $R(x)$ of degree ≤ 31.
3. The coefficients of $R(x)$ are considered to be a 32-bit sequence.
4. The bit sequence is complemented and the result is the CRC.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of a file to be checked. If no *file* operands are specified, the standard input is used.

STDIN

The standard input is used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cksum*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
For each file processed successfully, the *cksum* utility will write in the following format:

```
"%u %d %s\n", <checksum>, <# of octets>, <pathname>
```

If no *file* operand was specified, the pathname and its leading space will be omitted.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All files were processed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The *cksum* utility is typically used to quickly compare a suspect file against a trusted version of the same, such as to ensure that files transmitted over noisy media arrive intact. However, this comparison cannot be considered cryptographically secure. The chances of a damaged file producing the same CRC as the original are astronomically small; deliberate deception is difficult, but probably not impossible.

Although input files to *cksum* can be any type, the results need not be what would be expected on character special device files or on file types not described by the **XSH** specification. Since this document does not specify the block size used when doing input, checksums of character special files need not process all of the data in those files.

The algorithm is expressed in terms of a bitstream divided into octets. If a file is transmitted between two systems and undergoes any data transformation (such as moving 8-bit characters into 9-bit bytes or changing ‘‘Little Endian’’ byte ordering to ‘‘Big Endian’’), identical CRC values cannot be expected. Implementations performing such transformations may extend *cksum* to handle such situations.

EXAMPLES
None.

FUTURE DIRECTIONS

None.

SEE ALSO

sum.

CHANGE HISTORY

First released in Issue 4.

NAME

cmp – compare two files

SYNOPSIS

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

DESCRIPTION

The *cmp* utility compares two files. The *cmp* utility will write no output if the files are the same. Under default options, if they differ, it will write to standard output the byte and line number at which the first difference occurred. Bytes and lines will be numbered beginning with 1.

OPTIONS

The *cmp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–l (Lower-case ell.) Write the byte number (decimal) and the differing bytes (octal) for each difference.

–s Write nothing for differing files; return exit status only.

OPERANDS

The following operands are supported:

file1 A pathname of the first file to be compared. If *file1* is –, the standard input will be used.

file2 A pathname of the second file to be compared. If *file2* is –, the standard input will be used.

If both *file1* and *file2* refer to standard input or refer to the same FIFO special, block special or character special file, the results are undefined.

STDIN

The standard input will be used only if the *file1* or *file2* operand refers to standard input. See **INPUT FILES**.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cmp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
In the POSIX locale, results of the comparison will be written to standard output. When no options are used, the format will be:

```
"%s %s differ: char %d, line %d\n", file1, file2, <byte number>,
<line number>
```

When the **–l** option is used, the format is:

```
"%d %o %o\n", <byte number>, <differing byte>, <differing byte>
```

for each byte that differs. The first <*differing byte*> number is from *file1* while the second is from *file2*. In both cases, <*byte number*> is relative to the beginning of the file, beginning with 1.

No output will be written to standard output when the **–s** option is used.

STDERR
Used only for diagnostic messages. If *file1* and *file2* are identical for the entire length of the shorter file, in the POSIX locale the following diagnostic message will be written, unless the **–s** option is specified:

```
"cmp: EOF on %s%s\n", <name of shorter file>, <additional info>
```

The <*additional info*> field is either null or a string that starts with a blank character and contains no newline characters. Some systems report on the number of lines in this case.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The files are identical.
1 The files are different; this includes the case where one file is identical to the first part of the other.
>1 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Although input files to *cmp* can be any type, the results might not be what would be expected on character special device files or on file types not described by the **XSH** specification. Since this document does not specify the block size used when doing input, comparisons of character special files need not compare all of the data in those files.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO

comm, *diff*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

col – filter reverse line-feeds (**TO BE WITHDRAWN**)

SYNOPSIS

EX `col [-bfpx]`

DESCRIPTION

The *col* utility reads from the standard input and writes to the standard output. It performs the line overlays implied by reverse line-feeds, and by forward and reverse half-line-feeds. Unless **–x** is used, all blank characters in the input will be converted to tab characters wherever possible.

The ASCII control characters SO and SI are assumed by *col* to start and end text in an alternative 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 written in the correct character set.

On input, the only control characters accepted are space, backspace, tab, carriage-return and newline characters, SI, SO, VT, reverse line-feed, forward half-line-feed and reverse half-line-feed. The VT character is an alternative form of full reverse line-feed, included for compatibility with some earlier programs of this type. The only other characters to be copied to the output are those that are printable.

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

OPTIONS

The *col* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–b Assume that the output device in use is not capable of backspacing. In this case, if two or more characters are to appear in the same place, only the last one read will be output.

–f Suppress the normal treatment of half-line motions. 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. By suppressing this treatment, the output from *col* may contain forward half-line-feeds, but will still never contain either kind of reverse-line motion.

–p Force escape sequences to be passed through unchanged. Normally, *col* will remove any escape sequences found in its input that are not specified above.

–x Prevent *col* from converting blank characters to tab characters on output wherever possible. Tab stops are considered to be at each column position *n* such that *n* modulo 8 equals 1.

OPERANDS

None.

STDIN

The standard input is a text file to be translated.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *col*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file, translated from the standard input.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The use of the **–x** option may increase or decrease printing time, depending on the printer type.

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.

The use of the **–f** or **–p** options is discouraged unless the user is aware of the consequences of passing unusual escape sequences to the terminal.

EXAMPLES

None.

FUTURE DIRECTIONS

The *col* utility will be withdrawn from a future issue. It may be replaced in a future issue by a similar utility without the current ASCII bias.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

comm – select or reject lines common to two files

SYNOPSIS

`comm [-123] file1 file2`

DESCRIPTION

The *comm* utility will read *file1* and *file2*, which should be ordered in the current collating sequence, and produce three text columns as output: lines only in *file1*; lines only in *file2*; and lines in both files.

If the lines in both files are not ordered according to the collating sequence of the current locale, the results are unspecified.

OPTIONS

The *comm* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–1 Suppress the output column of lines unique to *file1*.

–2 Suppress the output column of lines unique to *file2*.

–3 Suppress the output column of lines duplicated in *file1* and *file2*.

OPERANDS

The following operands are supported:

file1 A pathname of the first file to be compared. If *file1* is –, the standard input is used.

file2 A pathname of the second file to be compared. If *file2* is –, the standard input is used.

If both *file1* and *file2* refer to standard input or to the same FIFO special, block special or character special file, the results are undefined.

STDIN

The standard input will be used only if one of the *file1* or *file2* operands refers to standard input. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *comm*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the collating sequence *comm* expects to have been used when the input files were sorted.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *comm* utility will produce output depending on the options selected. If the **–1**, **–2** and **–3** options are all selected, *comm* will write nothing to standard output.

If the **–1** option is not selected, lines contained only in *file1* will be written using the format:

```
"%s\n", <line in file1>
```

If the **–2** option is not selected, lines contained only in *file2* will be written using the format:

```
"%s%s\n", <lead>, <line in file2>
```

where the string <*lead*> is:

<tab> if the **–1** option is not selected, or

null string if the **–1** option is selected.

If the **–3** option is not selected, lines contained in both files will be written using the format:

```
"%s%s\n", <lead>, <line in both>
```

where the string <*lead*> is:

<tab><tab> if neither the **–1** nor the **–2** option is selected, or

<tab> if exactly one of the **–1** and **–2** options is selected, or

null string if both the **–1** and **–2** options are selected.

If the input files were ordered according to the collating sequence of the current locale, the lines written will be in the collating sequence of the original lines.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All input files were successfully output as specified.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
If the input files are not properly presorted, the output of *comm* might not be useful.

EXAMPLES

If a file named **xpg4** contains a sorted list of the utilities in this document, a file named **xpg3** contains a sorted list of the utilities specified in the **X/Open Portability Guide**, **Issue 3**, and a file named **svid89** contains a sorted list of the utilities in the System V Interface Definition Third Edition:

```
comm -23 xpg4 xpg3 | comm -23 - svid89
```

would print a list of utilities in this document not specified by either of the other documents;

```
comm -12 xpg4 xpg3 | comm -12 - svid89
```

would print a list of utilities specified by all three documents; and

```
comm -12 xpg3 svid89 | comm -23 - xpg4
```

would print a list of utilities specified by both XPG3 and the SVID, but not specified in this document.

FUTURE DIRECTIONS

None.

SEE ALSO

cmp, *diff*, *sort*, *uniq*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

command – execute a simple command

SYNOPSIS

```
command [-p] ommand_name [argument ...]
command [ -v | -V ] command_name
```

DESCRIPTION

The *command* utility causes the shell to treat the arguments as a simple command, suppressing the shell function lookup that is described in **Command Search and Execution** on page 47 item 1b.

If the *command_name* is the same as the name of one of the special built-in utilities, the special properties in the enumerated list at the beginning of Section 2.14 on page 67 will not occur. In every other respect, if *command_name* is not the name of a function, the effect of *command* will be the same as omitting *command*.

The *command* utility also provides information concerning how a command name will be interpreted by the shell; see **–v** and **–V**.

OPTIONS

The *command* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–p Perform the command search using a default value for *PATH* that is guaranteed to find all of the standard utilities.

–v Write a string to standard output that indicates the pathname or command that will be used by the shell, in the current shell execution environment (see Section 2.12 on page 63), to invoke *command_name*.

- Utilities, regular built-in utilities, *command_names* including a slash character, and any implementation-provided functions that are found using the *PATH* variable (as described in **Command Search and Execution** on page 47), will be written as absolute pathnames.
- Shell functions, special built-in utilities, regular built-in utilities not associated with a *PATH* search, and shell reserved words will be written as just their names.
- An alias will be written as a command line that represents its alias definition.
- Otherwise, no output will be written and the exit status will reflect that the name was not found.

–V Write a string to standard output that indicates how the name given in the *command_name* operand will be interpreted by the shell, in the current shell execution environment (see Section 2.12 on page 63). Although the format of this string is unspecified, it will indicate in which of the following categories *command_name* falls and include the information stated:

- Utilities, regular built-in utilities, and any implementation-provided functions that are found using the *PATH* variable (as described in **Command Search and Execution** on page 47), will be identified as such and include the absolute pathname in the string.
- Other shell functions will be identified as functions.
- Aliases will be identified as aliases and their definitions will be included in the string.

- Special built-in utilities will be identified as special built-in utilities.
- Regular built-in utilities not associated with a *PATH* search will be identified as regular built-in utilities. (The term ‘‘regular’’ need not be used.)
- Shell reserved words will be identified as reserved words.

OPERANDS

The following operands are supported:

argument
: One of the strings treated as an argument to *command_name*.

command_name
: The name of a utility or a special built-in utility.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *command*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path used during the command search described in **Command Search and Execution** on page 47, except as described under the **–p** option.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–v** option is specified, standard output is formatted as:

```
"%s\n", <pathname or command>
```

When the **–V** option is specified, standard output is formatted as:

```
"%s\n", <unspecified>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

When the **–v** or **–V** options are specified, the following exit values are returned:

0 Successful completion.
>0 The *command_name* could not be found or an error occurred.

Otherwise, the following exit values are returned:

126 The utility specified by *command_name* was found but could not be invoked.
127 An error occurred in the *command* utility or the utility specified by *command_name* could not be found.

Otherwise, the exit status of *command* will be that of the simple command specified by the arguments to *command*.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The order for command search allows functions to override regular built-ins and path searches. This utility is necessary to allow functions that have the same name as a utility to call the utility (instead of a recursive call to the function).

The system default path is available using *getconf*; however, since *getconf* may need to have the *PATH* set up before it can be called itself, the following can be used:

```
command -p getconf _CS_PATH
```

There are some advantages to suppressing the special characteristics of special built-ins on occasion. For example:

```
command exec > unwritable-file
```

will not cause a non-interactive script to abort, so that the output status can be checked by the script.

The *command*, *env*, *nohup*, *time* and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

Since the **–v** and **–V** options of *command* produce output in relation to the current shell execution environment, *command* is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(PATH=foo command -v)
nohup command -v
```

it will not necessarily produce correct results. For example, when called with *nohup* or an *exec* function, in a separate utility execution environment, most implementations will not be able to identify aliases, functions or special built-ins.

Two types of regular built-ins could be encountered on a system and these are described separately by *command*. The description of command search in **Command Search and Execution** on page 47 allows for a standard utility to be implemented as a regular built-in as long as it is found in the appropriate place in a *PATH* search. So, for example, *command* **–v** true might yield **/bin/true** or some similar pathname. Other implementation-provided utilities that are not defined by this document might exist only as built-ins and have no pathname associated with them. These will produce output identified as (regular) built-ins. Applications encountering these will not be able to count on *exec*ing them, using them with *nohup*, overriding them with a different *PATH*, and so forth.

EXAMPLES

1. Make a version of *cd* that always prints out the new working directory exactly once:

```
cd() {
        command cd "$@" >/dev/null
        pwd
}
```

2. Start off a ‘‘secure shell script’’ in which the script avoids being spoofed by its parent:

```
IFS='
'
#       The preceding value should be <space><tab><newline>.
#       Set IFS to its default value.

\unalias -a
#       Unset all possible aliases.
#       Note that unalias is escaped to prevent an alias
#       being used for unalias.

unset -f command
#       Ensure command is not a user function.

PATH="$(command -p getconf _CS_PATH):$PATH"
#       Put on a reliable PATH prefix.

#       ...
```

   At this point, given correct permissions on the directories called by *PATH*, the script has the ability to ensure that any utility it calls is the intended one. It is being very cautious because it assumes that implementation extensions may be present that would allow user functions to exist when it is invoked; this capability is not specified by this document, but it is not prohibited as an extension. For example, the *ENV* variable precedes the invocation of the script with a user startup script. Such a script could define functions to spoof the application.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, *type*.

CHANGE HISTORY

First released in Issue 4.

NAME

compress – compress data

SYNOPSIS

EX `compress [-fv][-b bits][file ...]`

EX `compress [-cfv][-b bits][file]`

DESCRIPTION

The *compress* utility will attempt to reduce the size of the named files by using adaptive Lempel-Ziv coding. Except when the output is to the standard output, each file will be replaced by one with the extension .Z. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file are preserved. If appending the .Z to the filename would make the name exceed {NAME_MAX} bytes, the command will fail. If no files are specified, the standard input will be compressed to the standard output.

OPTIONS

The *compress* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**b** *bits* Specify the maximum number of bits to use in a code. For a portable application, the *bits* argument must be:

$$9 \geq bits \leq 14$$

The implementation may allow *bits* values of greater than 14. The default will be 14, 15 or 16.

–**c** Cause *compress* to write to the standard output; the input file will not be changed, and no .Z files will be created.

–**f** Force compression of *file*, even if it does not actually reduce the size of the file, or if the corresponding *file*.Z file already exists. If the –**f** option is not given, and the process is not running in the background, the user will be prompted as to whether an existing *file*.Z file should be overwritten.

–**v** Write the percentage reduction of each file to standard error.

OPERANDS

The following operand is supported:

file A pathname of a file to be compressed.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –.

INPUT FILES

If *file* operands are specified, the input files contain the data to be compressed.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *compress*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If no *file* operands are specified, or if a *file* operand is –, or if the **–c** option is specified, the standard output will contain the compressed output.

STDERR
Used for all diagnostic and prompt messages and the output from **–v**.

OUTPUT FILES
The output files will contain the compressed output.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
1 An error occurred.
2 One or more files were not compressed because they would have increased in size (and the **–f** option was not specified).
>2 An error occurred.

CONSEQUENCES OF ERRORS
The input file will remain unmodified.

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

Although *compress* strictly follows the default actions upon receipt of a signal or when an error occurs, some unexpected results may occur. In some implementations it is likely that a partially compressed file will be left in place, alongside its uncompressed input file. Since the general operation of *compress* is to delete the uncompressed file only after the .Z file has been successfully filled, an application should always carefully check the exit status of *compress* before arbitrarily deleting files that have like-named neighbours with .Z suffixes.

Compressed files are not necessarily portable to other systems.

The limit of 14 on the *bits* option-argument is to achieve portability to all systems (within the restrictions imposed by the lack of an explicit published file format). Some systems based on 16-bit architectures cannot support 15- or 16-bit uncompression.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

pack, *uncompress*, *zcat*.

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

The following changes are made:

- The **DESCRIPTION** is clarified to state that the ownership, modes, access time, and modification time of the original file are preserved if the invoking process has appropriate privileges.
- The **STDOUT** section includes the case where a *file* operand is –.

NAME

cp – copy files

SYNOPSIS

```
cp [-fip] source_file target_file
cp [-fip] source_file ... target
cp -R[-fip] source_file ... target
cp -r[-fip] source_file ... target
```

DESCRIPTION

The first synopsis form is denoted by two operands, neither of which are existing files of type directory. The *cp* utility will copy the contents of *source_file* to the destination path named by *target_file.*

The second synopsis form is denoted by two or more operands where the **–R** or **–r** options are not specified and the first synopsis form is not applicable. It is an error if any *source_file* is a file of type directory, if *target* does not exist or if *target* is a file of a type defined by the **XSH** specification, but is not a file of type directory. The *cp* utility will copy the contents of each *source_file* to the destination path named by the concatenation of *target,* a slash character and the last component of *source_file.*

The third and fourth synopsis forms are denoted by two or more operands where the **–R** or **–r** options are specified. The *cp* utility will copy each file in the file hierarchy rooted in each *source_file* to a destination path named as follows.

If *target* exists and is a file of type directory, the name of the corresponding destination path for each file in the file hierarchy will be the concatenation of *target,* a slash character and the pathname of the file relative to the directory containing *source_file.*

If *target* does not exist and two operands are specified, the name of the corresponding destination path for *source_file* will be *target;* the name of the corresponding destination path for all other files in the file hierarchy will be the concatenation of *target,* a slash character and the pathname of the file relative to *source_file.*

It is an error if *target* does not exist and more than two operands are specified, or if *target* exists and is a file of a type defined by the **XSH** specification, but is not a file of type directory.

In the following description, *source_file* refers to the file that is being copied, whether specified as an operand or a file in a file hierarchy rooted in a *source_file* operand. The term *dest_file* refers to the file named by the destination path.

For each *source_file,* the following steps will be taken:

1. If *source_file* references the same file as *dest_file, cp* may write a diagnostic message to standard error; it will do nothing more with *source_file* and will go on to any remaining files.
2. If *source_file* is of type directory, the following steps will be taken:
   a. If neither the **–R** or **–r** options were specified, *cp* will write a diagnostic message to standard error, do nothing more with *source_file* and go on to any remaining files.
   b. If *source_file* was not specified as an operand and *source_file* is dot or dot-dot, *cp* will do nothing more with *source_file* and go on to any remaining files.
   c. If *dest_file* exists and it is a file type not specified by the **XSH** specification, the behaviour is implementation-dependent.

d. If *dest_file* exists and it is not of type directory, *cp* will write a diagnostic message to standard error, do nothing more with *source_file* or any files below *source_file* in the file hierarchy, and go on to any remaining files.

e. If the directory *dest_file* does not exist, it will be created with file permission bits set to the same value as those of *source_file*, modified by the file creation mask of the user if the **–p** option was not specified, and then bitwise inclusively ORed with S_IRWXU. If *dest_file* cannot be created, *cp* will write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files. It is unspecified if *cp* will attempt to copy files in the file hierarchy rooted in *source_file*.

f. The files in the directory *source_file* will be copied to the directory *dest_file*, taking the four steps [1–4] listed here with the files as *source_file*s.

g. If *dest_file* was created, its file permission bits will be changed (if necessary) to be the same as those of *source_file*, modified by the file creation mask of the user if the **–p** option was not specified.

h. The *cp* utility will do nothing more with *source_file* and go on to any remaining files.

3. If *source_file* is of type regular file, the following steps will be taken:

   a. If *dest_file* exists, the following steps are taken:

      i. If the **–i** option is in effect, the *cp* utility will write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, *cp* will do nothing more with *source_file* and go on to any remaining files.

      ii. A file descriptor for *dest_file* will be obtained by performing actions equivalent to the **XSH** specification *open*( ) function called using *dest_file* as the *path* argument, and the bitwise inclusive OR of O_WRONLY and O_TRUNC as the *oflag* argument.

      iii. If the attempt to obtain a file descriptor fails and the **–f** option is in effect, *cp* will attempt to remove the file by performing actions equivalent to the **XSH** specification *unlink*( ) function called using *dest_file* as the *path* argument. If this attempt succeeds, *cp* will continue with step 3b.

   b. If *dest_file* does not exist, a file descriptor will be obtained by performing actions equivalent to the **XSH** specification *open*( ) function called using *dest_file* as the *path* argument, and the bitwise inclusive OR of O_WRONLY and O_CREAT as the *oflag* argument. The file permission bits of *source_file* will be the *mode* argument.

   c. If the attempt to obtain a file descriptor fails, *cp* will write a diagnostic message to standard error, do nothing more with *source_file*, and go on to any remaining files.

   d. The contents of *source_file* will be written to the file descriptor. Any write errors will cause *cp* to write a diagnostic message to standard error and continue to step 3e.

   e. The file descriptor will be closed.

   f. The *cp* utility will do nothing more with *source_file*. If a write error occurred in step 3d, it is unspecified if *cp* continues with any remaining files. If no write error occurred in step 3d, *cp* will go on to any remaining files.

4. Otherwise, the following steps will be taken:

   a. If the **–r** option was specified, the behaviour is implementation-dependent.

b. If the –**R** option was specified, the following steps will be taken:

i. The *dest_file* will be created with the same file type as *source_file*.

ii. If *source_file* is a file of type FIFO, the file permission bits will be the same as those of *source_file*, modified by the file creation mask of the user if the –**p** option was not specified. Otherwise, the permissions, owner ID and group ID of *dest_file* are implementation-dependent.

If this creation fails for any reason, *cp* will write a diagnostic message to standard error, do nothing more with *source_file* and go on to any remaining files.

If the implementation provides additional or alternate access control mechanisms (see **file access permissions** in the **XBD** specification, **Chapter 2**, **Glossary**), their effect on copies of files is implementation-dependent.

OPTIONS

The *cp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**f** If a file descriptor for a destination file cannot be obtained, as described in step 3.a.ii., attempt to unlink the destination file and proceed.

–**i** Write a prompt to standard error before copying to any existing destination file. If the response from the standard input is affirmative, the copy will be attempted, otherwise not.

–**p** Duplicate the following characteristics of each source file in the corresponding destination file:

1. The time of last data modification and time of last access. If this duplication fails for any reason, *cp* will write a diagnostic message to standard error.
2. The user ID and group ID. If this duplication fails for any reason, it is unspecified whether *cp* writes a diagnostic message to standard error.
3. The file permission bits and the S_ISUID and S_ISGID bits. Other, implementation-dependent, bits may be duplicated as well. If this duplication fails for any reason, *cp* will write a diagnostic message to standard error.

If the user ID or the group ID cannot be duplicated, the file permission bits S_ISUID and S_ISGID will be cleared. If these bits are present in the source file but are not duplicated in the destination file, it is unspecified whether *cp* writes a diagnostic message to standard error.

The order in which the preceding characteristics are duplicated is unspecified. The *dest_file* will not be deleted if these characteristics cannot be preserved.

–**R** Copy file hierarchies.

–**r** Copy file hierarchies. The treatment of special files is implementation-dependent.

OPERANDS

The following operands are supported:

source_file

A pathname of a file to be copied.

target_file
A pathname of an existing or non-existing file, used for the output when a single file is copied.

target A pathname of a directory to contain the copied files.

STDIN

Used to read an input line in response to each prompt specified in **STDERR**. Otherwise, the standard input will not be used.

INPUT FILES

The input files specified as operands may be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES
Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

A prompt will be written to standard error under the conditions specified in **DESCRIPTION**. The prompt will contain the destination pathname, but its format is otherwise unspecified. Otherwise, the standard error will be used only for diagnostic messages.

OUTPUT FILES

The output files may be of any type.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files were copied successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If *cp* is prematurely terminated by a signal or error, files or file hierarchies may be only partially copied and files and directories may have incorrect permissions or access and modification times.

APPLICATION USAGE

The difference between **–R** and **–r** is in the treatment by *cp* of file types other than regular and directory. The original **–r** flag, for historic reasons, does not handle special files any differently from regular files, but always reads the file and copies its contents. This has obvious problems in the presence of special file types, for example character devices, FIFOs and sockets. The **–R** option is intended to recreate the file hierarchy and the **–r** option supports historical practice. It is anticipated that a future issue of this document will deprecate the **–r** option, and for that reason, there has been no attempt to fix its behaviour with respect to FIFOs or other file types where copying the file is clearly wrong. However, some systems support **–r** with the same abilities as the **–R** defined in the ISO / IEC 9945-2: 1993 standard. To accommodate them as well as systems that do not, the differences between **–r** and **–R** are implementation-dependent. Implementations may make them identical.

The set-user-ID and set-group-ID bits are explicitly cleared when files are created. This is to prevent users from creating programs that are set-user-ID or set-group-ID to them when copying files or to make set-user-ID or set-group-ID files accessible to new groups of users. For example, if a file is set-user-ID and the copy has a different group ID than the source, a new group of users has execute permission to a set-user-ID program than did previously. In particular, this is a problem for superusers copying users' trees.

EXAMPLES

None.

FUTURE DIRECTIONS

The **–r** option may be removed; use **–R** instead.

SEE ALSO

mv, *cpio*, *find*, *ln*, *pax*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

cpio – copy file archives in and out (**TO BE WITHDRAWN**)

SYNOPSIS

EX `cpio -o[aBcv]`

EX `cpio -i[Bcdmrtuvf] [pattern ...]`

EX `cpio -p[adlmuv] directory`

DESCRIPTION

The *cpio* utility, depending on the options used:

- copies files to an archive file
- extracts files from an archive file
- copies files from one directory tree to another.

OPTIONS

The *cpio* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except the option modifiers cannot be presented as separate arguments from the option letters.

The following options are supported:

–o (Copy Out.) Read the standard input to obtain a list of pathnames and copy those files onto the standard output together with pathname and status information. Output is padded to a 512-byte boundary.

–i (Copy In.) Extract files from the standard input, which is assumed to be the product of a previous *cpio* –**o**. Only files with names that match *pattern*s are selected. 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 has appropriate privileges, which causes *cpio* to retain the owner and group of the files of the previous *cpio* –**o**. If the archive being read does not match the modifier specified, *cpio* may consider this to be an error and exit or may recognise the archive and continue processing. Only a user with appropriate privileges can extract block special or character special files from an archive.

–p (Pass.) Read the standard input to obtain a list of pathnames of files that are conditionally created and copied into the destination *directory* tree based upon the option modifiers described below.

The following option modifiers can be appended in any sequence to the **–o**, **–i** or **–p** options:

a Reset access times of input files after they have been copied. (When option l (see below) is also specified, the access times of the linked files are not reset.)

B Block input/output 5120 bytes to the record (does not apply to the **–p** option; meaningful only with data directed to or from character special files).

d Create directories as needed.

c Write or read header information in character form for portability.

r Interactively rename files. For each archive member matching *pattern* operand, a prompt will be written to the file **/dev/tty**. The prompt will contain the name of the archive member, but the format is otherwise unspecified. A line will then be read from **/dev/tty**. If this line is blank, the archive member will be skipped. If this line consists of a single period, the archive member will be processed with no modification to its name.

Otherwise, its name will be replaced with the contents of the line. The *cpio* utility will immediately exit with a non-zero exit status if end-of-fie is encountered when reading a response, or if **/dev/tty** cannot be opened for reading and writing.

t Write 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: print the names of the affected files. 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 *pattern*s.

OPERANDS

The following operands are supported:

directory
A pathname of an existing directory to be used as the target of *cpio* **–p**.

pattern Expressions making use of a pattern-matching notation similar to that used by the shell for filename pattern matching, and similar to regular expressions. The following metacharacters are defined:

* Matches any string, including the empty string.

? Matches any single character.

[. . .] Matches any one of the enclosed characters. A pair of characters separated by '–' matches any symbol between the pair (inclusive), as defined by the system default collating sequence. If the first character following the opening '[' is a '!', the results are unspecified.

In *pattern*, the special characters ?, * and [also match the / character. Multiple cases of *pattern* can be specified and if no *pattern* is specified, the default for *pattern* is * (that is, select all files).

STDIN

When the **–o** or **–p** options are used, the standard input is a text file containing a list of pathnames, one per line, to be copied.

When the **–i** option is used, the standard input is an archive file formatted as specified by *pax* with the **–x cpio** option.

INPUT FILES

The files identified by the pathnames in the standard input are of any type.

When the **–r** option is used, the file **/dev/tty** is used to write prompts and read responses.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *cpio*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within bracketed filename patterns.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within bracketed filename patterns (for example, '[[:lower:]]*').

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format of date and time strings output when listing the contents of an archive with the –**v** option, for example:

```
cpio –icvt < /dev/sctmtm0
```

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When the –**o** option is used, the standard output is an archive file formatted as specified by *pax* with the –**x cpio** option. Otherwise, the standard output contains commentary in an unspecified format concerning the progress of the execution.

STDERR
When the –**o** option is not used, the standard error contains commentary in an unspecified format concerning the progress of the execution. Otherwise, the standard error is used only for diagnostic messages.

OUTPUT FILES
Output files are created, as specified by the archive, when the –**i** or –**p** options are used.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
If a file or directory cannot be created or overwritten, *cpio* continues with the next file in the archive or file to be added to the archive.

APPLICATION USAGE
Archives created by *cpio* are portable between XSI-conformant systems provided the same

procedures are used.

The shell metacharacter notation is not fully compatible with that used by the shell and the *pax* utility. Not all systems support the use of the negation character [! ...] in *cpio* patterns. Portable applications must avoid the use of this notation.

For portable communication of data between XSI-conformant systems, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the Portable Filename Character Set be used for naming files. This recommendation is given because XSI-conformant systems support diverse codesets and run in various geographical areas and there is no single, well-established codeset that incorporates all of the characters of the languages of the various geographical areas.

EXAMPLES

1. Copy the contents of a directory onto an archive:

```
ls | cpio –oc >../cpio.out
```

2. Duplicate a directory hierarchy:

```
cd olddir
find . –depth –print | cpio –pd ../newdir
```

FUTURE DIRECTIONS

This utility will be withdrawn from a future issue. The *pax* utility should be used instead. Modifications to achieve full conformance to the Utility Syntax Guidelines were not made.

SEE ALSO

ar, *find*, *ls*, *pax*, *tar*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

crontab – schedule periodic background work

SYNOPSIS

```
crontab [file]

crontab [ -e | -l | -r ]
```

DESCRIPTION

The *crontab* utility creates, replaces or edits a user's *crontab* entry; a crontab entry is a list of commands and the times at which they are to be executed. The new crontab entry can be input by specifying *file* or input from standard input if no *file* operand is specified, or by using an editor, if –**e** is specified.

Upon execution of a command from a crontab entry, the implementation will supply a default environment, defining at least the following environment variables:

HOME A pathname of the user's home directory.

LOGNAME
The user's login name.

PATH A string representing a search path guaranteed to find all of the standard utilities.

SHELL A pathname of the command interpreter. When *crontab* is invoked as specified by this document, the value will be a pathname for *sh*.

The values of these variables when *crontab* is invoked as specified by this document will not affect the default values provided when the scheduled command is run.

If standard output and standard error are not redirected by commands executed from the crontab entry, any generated output or errors will be mailed, via an implementation-dependent method, to the user.

EX 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 a process with appropriate privileges is allowed to submit a job. If only **cron.deny** exists and is empty, global usage is permitted. The **cron.allow** and **cron.deny** files consist of one user name per line.

OPTIONS

The *crontab* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**e** Edit a copy of the invoking user's crontab entry, or create an empty entry to edit if the crontab entry does not exist. When editing is complete, the entry will be installed as the user's crontab entry.

–**l** (The letter ell.) List the invoking user's crontab entry.

–**r** Remove the invoking user's crontab entry.

OPERANDS

The following operand is supported:

file The pathname of a file that contains specifications, in the format defined in **INPUT FILES**, for crontab entries.

STDIN

See **INPUT FILES**.

INPUT FILES

In the POSIX locale, a crontab entry must be a text file consisting of lines of six fields each. The fields must be separated by blank characters. The first five fields must be integer patterns that specify the following:

1. Minute (0–59)
2. Hour (0–23)
3. Day of the month (1–31)
4. Month of the year (1–12)
5. Day of the week (0–6 with 0=Sunday).

Each of these patterns can be either an asterisk (meaning all valid values), an element or a list of elements separated by commas. An element must be either a number or two numbers separated by a hyphen (meaning an inclusive range). The specification of days can be made by two fields (day of the month and day of the week). If month, day of month and day of week are all asterisks, every day will be matched. If either the month or day of month is specified as an element or list, but the day of week is an asterisk, the month and day of month fields will specify the days that match. If both month and day of month are specified as asterisk, but day of week is an element or list, then only the specified days of the week will match. Finally, if either the month or day of month is specified as an element or list, and the day of week is also specified as an element or list, then any day matching either the month and day of month or the day of week, will be matched.

The sixth field of a line in a crontab entry is a string that will be executed by *sh* at the specified times. A percent sign character in this field will be translated to a newline character. Any character preceded by a backslash (including the %) causes that character to be treated literally. Only the first line (up to a % or end of line) of the command field will be executed by the command interpreter. The other lines will be made available to the command as standard input.

Blank lines and those whose first non-blank character is # will be ignored.

EX The text files **/usr/lib/cron/cron.allow** and **/usr/lib/cron/cron.deny** contain user names, one per line, of users who are, respectively, authorised or denied access to the service underlying the *crontab* utility.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *crontab*:

EDITOR
: Determine the editor to be invoked when the **–e** option is specified. The default editor is *vi*.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If the **–l** option is specified, the crontab entry will be written to the standard output.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
The user's crontab entry is not submitted, removed, edited or listed.

APPLICATION USAGE
The format of the *crontab* entry shown here is guaranteed only for the POSIX locale. Other cultures may be supported with substantially different interfaces, although implementations are encouraged to provide comparable levels of functionality.

The default settings of the *HOME*, *LOGNAME*, *PATH* and *SHELL* variables that are given to the scheduled job are not affected by the settings of those variables when *crontab* is run; as stated, they are defaults. The text about ''invoked as specified by this document'' means that the implementation may provide extensions that allow these variables to be affected at runtime, but that the user has to take explicit action in order to access the extension, such as give a new option flag or modify the format of the crontab entry.

A typical user error is to type only *crontab*; this will cause the system to wait for the new crontab entry on standard input. If end-of-file is typed (generally <control>-D), the crontab entry will be replaced by an empty file. In this case, the user should type the interrupt character, which will prevent the crontab entry from being replaced.

EXAMPLES

1. Clean up **core** files every weekday morning at 3:15 am:

```
15 3 * * 1-5 find $HOME -name core 2>/dev/null | xargs rm -f
```

2. Mail a birthday greeting:

```
0 12 14 2 * mailx john%Happy Birthday!%Time for lunch.
```

3. As an example of specifying the two types of days:

```
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.

FUTURE DIRECTIONS

None.

SEE ALSO

at.

CHANGE HISTORY

First released in Issue 2.

Issue 3

References to ‘‘superuser’’ have been changed to ‘‘process with appropriate privileges’’; otherwise, functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

csplit – split files based on context

SYNOPSIS

`csplit [-ks][-f prefix][-n number] file arg1 ...argn`

DESCRIPTION

The *csplit* utility reads the file named by the *file* operand, writes all or part of that file into other files as directed by the *arg* operands, and writes the sizes of the files.

OPTIONS

The *csplit* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–f *prefix*
: Name the created files *prefix***00**, *prefix***01**, ..., *prefix**n*. The default is **xx00** ... **xx***n*. If the *prefix* argument would create a filename exceeding {NAME_MAX} bytes, an error will result, *csplit* will exit with a diagnostic message and no files will be created.

–k
: Leave previously created files intact. By default, *csplit* will remove created files if an error occurs.

–n *number*
: Use *number* decimal digits to form filenames for the file pieces. The default is 2.

–s
: Suppress the output of file size messages.

OPERANDS

The following operands are supported:

file
: The pathname of a text file to be split. If *file* is –, the standard input will be used.

The operands *arg1* ... *argn* can be a combination of the following:

/*rexp*/**[***offset***]**
: Create a file using the content of the lines from the current line up to, but not including, the line that results from the evaluation of the regular expression with *offset*, if any, applied. The regular expression *rexp* must follow the rules for basic regular expressions described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**. The optional *offset* must be a positive or negative integer value representing a number of lines. The integer value must be preceded by + or –. If the selection of lines from an offset expression of this type would create a file with zero lines, or one with greater than the number of lines left in the input file, the results are unspecified. After the section is created, the current line will be set to the line that results from the evaluation of the regular expression with any offset applied. The pattern match of *rexp* always is applied from the current line to the end of the file.

%*rexp*%**[***offset***]**
: This operand is the same as /*rexp*/**[***offset***]**, except that no file will be created for the selected section of the input file.

line_no
: Create a file from the current line up to (but not including) the line number *line_no*. Lines in the file will be numbered starting at one. The current line becomes *line_no*.

{*num***}**
: Repeat operand. This operand can follow any of the operands described previously. If it follows a *rexp* type operand, that operand will be applied *num* more times. If it follows a *line_no* operand, the file will be split every *line_no* lines, *num* times, from that point.

An error will be reported if an operand does not reference a line between the current position and the end of the file.

STDIN

See **INPUT FILES**.

INPUT FILES

The input file must be a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *csplit*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within regular expressions.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If the **–k** option is specified, created files will be retained. Otherwise the default action occurs.

STDOUT

Unless the **–s** option is used, the standard output will consist of one line per file created, with a format as follows:

```
"%d\n", <file size in bytes>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files will contain portions of the original input file, otherwise unchanged.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

By default, created files will be removed if an error occurs. When the **–k** option is specified, created files will not be removed if an error occurs.

APPLICATION USAGE

None.

EXAMPLES

1. This example creates four files, **cobol00** ... **cobol03**:

   ```
   csplit –f cobol 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.

2. This example would split the file after the first 99 lines, and every 100 lines there after, up to 9 999 lines; this is because lines in the file are numbered from 1 rather than zero, for historical reasons.

   ```
   csplit –k file  100  {99}
   ```

3. Assuming that **prog.c** follows the C-language 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**:

   ```
   csplit –k prog.c '%main(%'  '/^}/+1' {20}
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

sed, *split*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

ctags – create a tags file (**DEVELOPMENT**) (**FORTRAN**)

SYNOPSIS

```
ctags [-a][-f tagsfile] pathname ...
ctags -x pathname ...
```

DESCRIPTION

The *ctags* utility creates a *tags* file or an index of objects from C-language or FORTRAN source files specified by the *pathname* operands. (FORTRAN source is processed only on systems supporting the **FORTRAN** option.) The tags file lists the locators of language-specific objects within the source files. A locator consists of a name, pathname and either basic regular expression or a line number that can be used in searching for the object definition. The objects that will be recognised are specified in **EXTENDED DESCRIPTION**.

OPTIONS

The *ctags* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Append to tags file.

–f *tagsfile*
Write the object locator lists into *tagsfile* instead of the default file named **tags** in the current directory.

–x Produce a list of object names, the line number and filename in which each is defined, as well as the text of that line, and write this to the standard output. A **tags** file is not created when **–x** is specified.

OPERANDS

The following *pathname* operands are supported:

*file***.c** Files with basenames ending with the **.c** suffix are treated as C-language source code. Such files that are not valid input to *c89* produce unspecified results.

*file***.h** Files with basenames ending with the **.h** suffix are treated as C-language source code. Such files that are not valid input to *c89* produce unspecified results.

*file***.f** Files with basenames ending with the **.f** suffix are treated as FORTRAN-language source code. Such files that are not valid input to *fort77* produce unspecified results.

The handling of other files is implementation-dependent.

STDIN

See **INPUT FILES**.

INPUT FILES

The input files must be text files containing source code in the language indicated by the operand filename suffixes.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ctags*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the order in which output is sorted for the **–x** option. The POSIX locale determines the order in which the tags file is written.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files). When processing C-language source code, if the locale is not compatible with the C locale described by the ISO C standard, the results are unspecified.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The list of object name information produced by the **–x** option is written to standard output in the following format:

```
"%s %d %s %s", <object-name>, <line-number>, <filename>, <text>
```

where <*text*> is the text of line <*line-number*> of file <*filename*>.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
When the **–x** option is not specified, the format of the output file is:

```
"%s\t%s\t/%s/\n", <identifier>, <filename>, <rexp>
```

where <*rexp*> is a basic regular expression (see the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**) that could be used by an editor to find the defining instance of <*identifier*> in <*filename*> (where ‘‘defining instance’’ is indicated by the declarations listed in **EXTENDED DESCRIPTION**).

An alternative format is:

```
"%s\t%s\t%d\n", <identifier>, <filename>, <lineno>
```

where <*lineno*> is a decimal line number that could be used by an editor to find <*identifier*> in <*filename*>. This alternative format is not produced by *ctags* when it is used as described by this document, but the standard utilities that process tags files are able to process this format as well as the preceding one.

In either format, the file will be sorted by identifier, based on the collation sequence in the POSIX locale.

EXTENDED DESCRIPTION

If the operand identifies C-language source, the *ctags* utility will attempt to produce an output line for each of the following objects:

- function definitions
- type definitions
- macros with arguments.

It may also produce output for any of the following objects:

- function prototypes
- structures
- unions
- global variable definitions
- enumeration types
- macros without arguments
- **#define** statements
- **#line** statements.

Any **#if** and **#ifdef** statements will produce no output. The tag **main** is treated specially in C programs. The tag formed is created by prefixing **M** to the name of the file, with the trailing **.c**, and leading pathname components (if any) removed. This special treatment of **main** makes the use of *ctags* practical in directories with more than one program.

If the operand identifies FORTRAN source, the *ctags* utility will produce an output line for each function definition. It may also produce output for any of the following objects:

- subroutine definitions
- COMMON statements
- PARAMETER statements
- DATA and BLOCK DATA statements
- statement numbers.

On systems that do not support the **FORTRAN** option, *ctags* produces unspecified results for FORTRAN source code files. It writes to standard error a message identifying this condition and causes a non-zero exit status to be produced.

It is implementation-dependent what other objects (including duplicate identifiers) produce output.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The output with **–x** is meant to be a simple index that can be written out as an off-line readable function index. If the input files to *ctags* (such as **.c** files) were not created using the same locales as those in effect when *ctags* **–x** is run, results might not be as expected.

The description of C-language processing says ‘‘will attempt to’’ because the C language can be greatly confused, especially through the use of **#defines**, and this utility would be of no use if the real C preprocessor were run to identify them. The output from *ctags* may be fooled and incorrect for various constructs.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

c89, *fort77*, *vi*.

CHANGE HISTORY

First released in Issue 4.

NAME

cu – call another system

SYNOPSIS

UN EX `cu -n[-dht] [-o | -e ][-l line][-s speed]`

UN EX `cu [-dht] [-o | -e ][-l line][-s speed] telno`

UN EX `cu [-dht] [-o | -e ][-s speed] -l line`

UN EX `cu [-dht] [-o | -e ] systemname`

DESCRIPTION

The *cu* utility calls up another system. It manages an interactive conversation, with possible transfers of text files.

OPTIONS

The *cu* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–**s** *speed*
Specify the transmission speed. The default value is device-specific.

–**l** *line* Specify 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 written and the call will not be made. If the specified device is associated with an autodialler, a telephone number must be provided.

–**h** Emulate local echo, supporting calls to other computer systems that expect terminals to be set to half-duplex mode.

–**t** Dial a remote modem that has been set to auto-answer. Appropriate mapping of carriage-return to carriage-return-line-feed pairs is set.

–**d** Write diagnostic traces.

–**o** Designate that odd parity is to be generated for data sent to the remote system.

–**e** Designate that even parity is to be generated for data sent to the remote system.

–**n** Prompt the user to provide the telephone number to be dialled rather than taking it from the command line. This is for added security.

OPERANDS

The following operands are supported:

telno When using an automatic dialler, specifies the telephone number with equal signs for secondary dial tone or minus signs placed appropriately for delays of 4 seconds.

systemname
Specifies a *uucp* system name, which can be used rather than a telephone number; in this case, *cu* will obtain an appropriate direct line or telephone number from a system file.

STDIN

The standard input is used to accept data to write to the remote system; see **EXTENDED DESCRIPTION**.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cu*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input data).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

The *cu* utility takes the default action upon receipt of signals, with the exception of:

SIGHUP

Close the connection and terminate.

SIGINT Forward to the remote system.

SIGQUIT

Forward to the remote system.

SIGUSR1

Terminate the *cu* process without the normal connection closing sequence.

STDOUT

The standard output is used to display data to read from the remote system; see **EXTENDED DESCRIPTION**.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

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 system so that the buffer is not overrun. Lines beginning with ~ have special meanings.

The *transmit* process interprets the following user-initiated commands as:

~. Terminate the conversation.

~! Escape to an interactive command interpreter on the local system.

~!*command*
Execute the shell *command* on the local system.

~$*command*
Run the shell *command* locally and send its output to the remote system for execution.

~%cd Change the directory on the local system.

~%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 or **~%b**
Transmit a BREAK to the remote system.

~%nostop
Toggle between DC3/DC1 input control protocol and no input control. This is useful in case the remote system is one that 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* and *cat* 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*) 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:

```
$ uname
Z
$ ~[X]!uname
X
$ ~~[Y]!uname
Y
```

(The darker type indicates system-generated text. The bracketed system names are written by the system after it has recognised the ~! pair, but before it echoed the !.) 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.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>1 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the the **XBD** specification, **Chapter 9**, **General Terminal Interface** interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility will write an error message describing the problem and exit with a non-zero exit status.

EXAMPLES

1. To dial a system whose telephone number is 9 1 201 555 1212 using a device-specific speed (where dial tone is expected after the 9):

   ```
   cu 9=12015551212
   ```

2. To login to a system connected by a direct line:

   ```
   cu –l /dev/ttyXX
   ```

   or:

   ```
   cu –l ttyXX
   ```

3. To dial a system with the specific line and a specific speed:

   ```
   cu –s 1200 –l ttyXX
   ```

4. To dial a system using a specific line associated with an autodialler:

   ```
   cu –l culXX 9=12015551212
   ```

5. To use a system name:

   ```
   cu systemname
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

cat, *echo*, *stty*, *uname*, *uucp*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

Presence of the utility mandated, even on systems where no communications are available.

NAME

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

SYNOPSIS

```
cut -b list [-n] [file ..]
cut -c list [file ...]
cut -f list [-d delim][-s][file ...]
```

DESCRIPTION

The *cut* utility will cut out bytes (–**b** option), characters (–**c** option) or character-delimited fields (–**f** option) from each line in one or more files, concatenate them and write them to standard output.

OPTIONS

The *cut* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The option-argument *list* (see options –**b**, –**c** and –**f** below) must be a comma-separated list or blank-character-separated list of positive numbers and ranges. Ranges can be in three forms. The first is two positive numbers separated by a hyphen (*low–high*), which represents all fields from the first number to the second number. The second is a positive number preceded by a hyphen (*–high*), which represents all fields from field number 1 to that number. The third is a positive number followed by a hyphen (*low–*), which represents that number to the last field, inclusive. The elements in list can be repeated, can overlap and can be specified in any order.

The following options are supported:

–**b** *list* Cut based on a *list* of bytes. Each selected byte will be output unless the –**n** option is also specified. It is not an error to select bytes not present in the input line.

–**c** *list* Cut based on a *list* of characters. Each selected character is output. It is not an error to select characters not present in the input line.

–**d** *delim*
Set the field delimiter to the character *delim*. The default is the tab character.

–**f** *list* Cut based on a *list* of fields, assumed to be separated in the file by a delimiter character (see –**d**). Each selected field will be output. Output fields will be separated by a single occurrence of the field delimiter character. Lines with no field delimiters will be passed through intact, unless –**s** is specified. It is not an error to select fields not present in the input line.

–**n** Do not split characters. When specified with the –**b** option, each element in *list* of the form *low–high* (hyphen-separated numbers) will be modified as follows:

- If the byte selected by *low* is not the first byte of a character, *low* will be decremented to select the first byte of the character originally selected by *low*. If the byte selected by *high* is not the last byte of a character, *high* will be decremented to select the last byte of the character prior to the character originally selected by *high*, or zero if there is no prior character. If the resulting range element has *high* equal to zero or *low* greater than *high*, the list element will be dropped from *list* for that input line without causing an error.

Each element in *list* of the form *low–* will be treated as above with *high* set to the number of bytes in the current line, not including the terminating newline character. Each element in *list* of the form *–high* will be treated as above with *low* set to 1. Each element in *list* of the form *num* (a single number) will be treated as above with *low* set to *num* and *high* set to *num*.

–s Suppress lines with no delimiter characters, when used with the **–f** option. Unless specified, lines with no delimiters will be passed through untouched.

OPERANDS

The following operands are supported:

file A pathname of an input file. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files, except that line lengths are unlimited.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cut*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *cut* utility output will be a concatenation of the selected bytes, characters or fields (one of the following):

```
"%s\n", <concatenation of bytes>
"%s\n", <concatenation of characters>
"%s\n", <concatenation of fields and field delimiters>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All input files were output successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Earlier versions of the *cut* utility worked in an environment where bytes and characters were considered equivalent (modulo backspace and tab character processing in some implementations). In the extended world of multi-byte characters, the new **–b** option has been added. The **–n** option (used with **–b**) allows it to be used to act on bytes rounded to character boundaries. The algorithm specified for **–n** guarantees that:

```
cut -b 1-500 -n file > file1
cut -b 501- -n file > file2
```

will end up with all the characters in **file** appearing exactly once in **file1** or **file2**. (There is, however, a newline character in both **file1** and **file2** for each newline character in **file**.)

EXAMPLES

Examples of the option qualifier list:

1,4,7 Select the first, fourth and seventh bytes, characters, or fields and field delimiters.

1–3,8 Equivalent to 1,2,3,8.

–5,10 Equivalent to 1,2,3,4,5,10.

3– Equivalent to third to last, inclusive.

The *low–high* forms are not always equivalent when used with **–b** and **–n** and multi-byte characters. See the description of **–n**.

The following command:

```
cut -d : -f 1,6 /etc/passwd
```

reads the System V password file (user database) and produces lines of the form:

```
<user ID>:<home directory>
```

Most utilities in this document work on text files. The *cut* utility can be used to turn files with arbitrary line lengths into a set of text files containing the same data. The *paste* utility can be used to create (or recreate) files with arbitrary line lengths. For example, if **file** contains long lines:

```
cut -b 1-500 -n file > file1
cut -b 501- -n file > file2
```

creates **file1** (a text file) with lines no longer than 500 bytes (plus the newline character and **file2** that contains the remainder of the data from **file**. (Note that **file2** will not be a text file if there are lines in **file** that are longer than 500 + {LINE_MAX} bytes.) The original file can be recreated from **file1** and **file2** using the command:

```
paste -d "\0" file1 file2 > file
```

FUTURE DIRECTIONS

None.

SEE ALSO

grep, *paste*, Section 2.5 on page 27.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

cxref – generate C-language program cross-reference table (**DEVELOPMENT**)

SYNOPSIS

EX `cxref [-cs][-o file][-w num] [-D name[=def]]...[-I dir]...[-U dir]... file ...`

DESCRIPTION

The *cxref* utility analyses a collection of C-language *file*s and attempts to build a cross-reference table. Information from **#define** lines is included in the symbol table. A sorted listing is written to 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.

OPTIONS

The *cxref* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the order of the **–D**, **–I** and **–U** options (which are identical to their interpretation by *c89*) is significant. The following options are supported:

–c Write a combined cross-reference of all input files.

–w *num*
Format output no wider than *num* (decimal) columns. This option defaults to 80 if *num* is not specified or is less than 51.

–o *file* Direct output to named *file*.

–s Operate silently; do not print input filenames.

OPERANDS

The following operand is supported:

file A pathname of a C-language source file.

STDIN

Not used.

INPUT FILES

The input files are C-language source files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *cxref*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the ordering of the output.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output is used for the cross-reference listing, unless the **−o** option is used to select a different output file.

The format of standard output is unspecified, except that the following information is included:

- If the **−c** option is not specified, each portion of the listing starts with the name of the input file on a separate line.
- The name line is followed by a sorted list of symbols, each with its associated location pathname, the name of the function in which it appears (if it is not a function name itself), and line number references.
- Each line number may be preceded by an asterisk (*) flag, meaning that this is the declaring reference. Other single-character flags, with implementation-dependent meanings, may be included.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The output file named by the **−o** option is used instead of standard output.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>1 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
cc, *c89*.

CHANGE HISTORY
First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

date – write the date and time

SYNOPSIS

```
date [-u] [+format]
EX  date [-u] mmddhhmm[yy]
```

DESCRIPTION

EX The *date* utility writes the date and time to standard output or attempts to set the system date and time. By default, the current date and time will be written. If an operand beginning with + is specified, the output format of *date* will be controlled by the field descriptors and other text in the operand.

OPTIONS

The *date* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–u Perform operations as if the *TZ* environment variable was set to the string UTC0, or its equivalent historical value of GMT0. Otherwise, *date* will use the timezone indicated by the *TZ* environment variable or the system default if that variable is not set.

OPERANDS

The following operands are supported:

+format

When the format is specified, each field descriptor will be replaced in the standard output by its corresponding value. All other characters will be copied to the output without change. The output will always be terminated with a newline character.

Field Descriptors

%a Locale's abbreviated weekday name.
%A Locale's full weekday name.
%b Locale's abbreviated month name.
%B Locale's full month name.
%c Locale's appropriate date and time representation.
%C Century (a year divided by 100 and truncated to an integer) as a decimal number [00–99].
%d Day of the month as a decimal number [01–31].
%D Date in the format *mm/dd/yy*.
%e Day of the month as a decimal number [1–31] in a two-digit field with leading space character fill.
%h A synonym for **%b**.
%H Hour (24-hour clock) as a decimal number [00–23].
%I Hour (12-hour clock) as a decimal number [01–12].
%j Day of the year as a decimal number [001–366].
%m Month as a decimal number [01–12].
%M Minute as a decimal number [00–59].
%n A newline character.
%p Locale's equivalent of either AM or PM.
%r 12-hour clock time [01–12] using the AM/PM notation; in the POSIX locale, this will be equivalent to "%I:%M:%S %p".
%S Seconds as a decimal number [00–61].
%t A tab character.

%T 24-hour clock time [00–23] in the format *HH:MM:SS*.
%u Weekday as a decimal number [1 (Monday)–7].
%U Week of the year (Sunday as the first day of the week) as a decimal number [00–53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%V Week of the year (Monday as the first day of the week) as a decimal number [01–53]. If the week containing January 1 has four or more days in the new year, then it is considered week 1; otherwise, it is week 53 of the previous year, and the next week is week 1. (See the ISO 8601: 1988 standard.)
%w Weekday as a decimal number [0 (Sunday)–6].
%W Week of the year (Monday as the first day of the week) as a decimal number [00–53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x Locale's appropriate date representation.
%X Locale's appropriate time representation.
%y Year (offset from **%C**) as a decimal number [00–99].
%Y Year with century as a decimal number.
%Z Timezone name, or no characters if no timezone is determinable.
%% A percent sign character.

See the LC_TIME description in the **XBD** specification, **Chapter 5**, **Locale** for the field descriptor values in the POSIX locale.

Modified Field Descriptors

Some field descriptors can be modified by the E and O modifier characters to indicate a different format or specification as specified in the LC_TIME locale description (see the **XBD** specification, **Chapter 5**, **Locale**). If the corresponding keyword (see **era**, **era_year**, **era_d_fmt** and **alt_digits** in the **XBD** specification, **Chapter 5**, **Locale**) is not specified or not supported for the current locale, the unmodified field descriptor value will be used.

%Ec Locale's alternative appropriate date and time representation.

%EC The name of the base year (period) in the locale's alternative representation.

%Ex Locale's alternative date representation.

EX %EX Locale's alternative time representation.

%Ey Offset from **%EC** (year only) in the locale's alternative representation.

%EY Full alternative year representation.

%Od Day of month using the locale's alternative numeric symbols.

%Oe Day of month using the locale's alternative numeric symbols.

%OH Hour (24-hour clock) using the locale's alternative numeric symbols.

%OI Hour (12-hour clock) using the locale's alternative numeric symbols.

%Om Month using the locale's alternative numeric symbols.

%OM Minutes using the locale's alternative numeric symbols.

%OS Seconds using the locale's alternative numeric symbols.

%Ou Weekday as a number in the locale's alternative representation (Monday = 1).

%OU Week number of the year (Sunday as the first day of the week) using the locale's alternative numeric symbols.

%OV Week number of the year (Monday as the first day of the week, rules corresponding to **%V**), using the locale's alternative numeric symbols.

%Ow Weekday as a number in the locale's alternative representation (Sunday = 0).

%OW Week number of the year (Monday as the first day of the week) using the locale's alternative numeric symbols.

%Oy Year (offset from **%C**) in alternative representation.

EX *mmddhhmm*[*yy*]

Attempt to set the system date and time from the value given in the operand. This is only possible if the user has appropriate privileges and the system permits the setting of the system date and time. The first *mm* is the month (number); *dd* is the day (number); *hh* is the hour (number, 24-hour system); the second *mm* is the minute (number) and *yy* is the last two digits of the year and is optional. For example:

```
date 10080045
```

sets the date to October 8, 00:45. The current year is the default if *yy* is omitted.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *date*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME

Determine the format and contents of date and time strings written by *date*.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone in which the time and date are written, unless the **–u** option is specified. If the *TZ* variable is not set and the **–u** is not specified, an unspecified system default timezone is used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When no formatting operand is specified, the output in the POSIX locale is equivalent to specifying:

```
date "+%a %b %e %H:%M:%S %Z %Y"
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The date was written successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Field descriptors are of unspecified format when not in the POSIX locale. Some of them can contain newline characters in some locales, so it may be difficult to use the format shown in standard output for parsing the output of *date* in those locales.

The range of values for **%S** extends from 0 to 61 seconds to accommodate the occasional leap second or double leap second.

Although certain of the field descriptors in the POSIX locale (such as the name of the month) are shown with initial capital letters, this need not be the case in other locales. Programs using these fields may need to adjust the capitalisation if the output is going to be used at the beginning of a sentence.

The date string formatting capabilities are intended for use in Gregorian style calendars, possibly with a different starting year (or years). The **%x** and **%c** field descriptors, however, are intended for local representation; these may be based on a different, non-Gregorian calendar.

The **%C** field descriptor was introduced to allow a fallback for the **%EC** (alternative year format base year); it can be viewed as the base of the current subdivision in the Gregorian calendar. A century is not calculated as an ordinal number; this Guide was published in century 19, not the twentieth. Both the **%Ey** and **%y** can then be viewed as the offset from **%EC** and **%C**, respectively.

The E and O modifiers modify the traditional field descriptors, so that they can always be used, even if the implementation (or the current locale) does not support the modifier.

The E modifier supports alternative date formats, such as the Japanese Emperor's Era, as long as these are based on the Gregorian calendar system. Extending the E modifiers to other date elements may provide an implementation-specific extension capable of supporting other calendar systems, especially in combination with the O modifier.

The O modifier supports time and date formats using the locale's alternative numerical symbols, such as Kanji or Hindi digits or ordinal number representation.

Non-European locales, whether they use Latin digits in computational items or not, often have local forms of the digits for use in date formats. This is not totally unknown even in Europe; a

variant of dates uses Roman numerals for the months: the third day of September 1991 would be written as 3.IX.1991. In Japan, Kanji digits are regularly used for dates; in Arabic-speaking countries, Hindi digits are used. The **%d**, **%e**, **%H**, **%I**, **%m**, **%S**, **%U**, **%w**, **%W** and **%y** field descriptors always return the date and time field in Latin digits (that is, 0 to 9). The **%O** modifier was introduced to support the use for display purposes of non-Latin digits. In the LC_TIME category in *localedef*, the optional **alt_digits** keyword is intended for this purpose. As an example, assume the following (partial) *localedef* source:

```
alt_digits    "";"I";"II";"III";"IV";"V";"VI";"VII";"VIII" \
              "IX";"X";"XI";"XII"
d_fmt         "%e.%Om.%Y"
```

With the above date, the command:

```
date "+x"
```

would yield 3.IX.1991. With the same **d_fmt**, but without the **alt_digits**, the command would yield 3.9.1991.

EXAMPLES

1. The following are input/output examples of *date* used at arbitrary times in the POSIX locale:

   ```
   $ date
   Tue Jun 26 09:58:10 PDT 1990

   $ date "+DATE: %m/%d/%y%nTIME: %H:%M:%S"
   DATE: 11/02/91
   TIME: 13:36:16

   $ date "+TIME: %r"
   TIME: 01:36:32 PM
   ```

2. Examples for Denmark, where the default date and time format is "%a %d %b %Y %T %Z":

   ```
   $ LANG=da_DK.iso_8859-1 date
   ons 02 okt 1991 15:03:32 CET

   $ LANG=da_DK.iso_8859-1 date "+DATO: %A den %e. %B %Y%nKLOKKEN: %H:%M:%S"
   DATO: onsdag den  2. oktober 1991
   KLOKKEN: 15:03:56
   ```

3. Examples for Germany, where the default date and time format is "%a %d.%h.%Y, %T %Z":

   ```
   $ LANG=De_DE.88591 date
   Mi 02.Okt.1991, 15:01:21 MEZ

   $ LANG=De_DE.88591 date "+DATUM: %A, %d. %B %Y%nZEIT:  %H:%M:%S"
   DATUM: Mittwoch, 02. Oktober 1991
   ZEIT:  15:02:02
   ```

4. Examples for France, where the default date and time format is "%a %d %h %Y %Z %T":

```
$ LANG=Fr_FR.88591 date
Mer 02 oct 1991 MET 15:03:32

$ LANG=Fr_FR.88591 date "+JOUR:  %A %d %B %Y%nHEURE: %H:%M:%S"
JOUR:  Mercredi 02 octobre 1991
HEURE: 15:03:56
```

FUTURE DIRECTIONS

None.

SEE ALSO

The **XSH** specification description of *ctime*( ), *printf*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

dd – convert and copy a file

SYNOPSIS

dd **[***operand* **...]**

DESCRIPTION

The *dd* utility will copy the specified input file to the specified output file with possible conversions using specific input and output block sizes. It will read the input one block at a time, using the specified input block size; it then will process the block of data actually returned, which could be smaller than the requested block size. It will apply any conversions that have been specified and write the resulting data to the output in blocks of the specified output block size. If the **bs=***expr* operand is specified and no conversions other than **sync**, **noerror** or **notrunc** are requested, the data returned from each input block will be written as a separate output block; if the read returns less than a full block and the **sync** conversion is not specified, the resulting output block will be the same size as the input block. If the **bs=***expr* operand is not specified, or a conversion other than **sync**, **noerror** or notrunc is requested, the input will be processed and collected into full-sized output blocks until the end of the input is reached.

The processing order is as follows:

1. An input block is read.
2. If the input block is shorter than the specified input block size and the **sync** conversion is specified, null bytes are appended to the input data up to the specified size. (If either **block** or **unblock** is also specified, space characters are appended instead of null bytes.) The remaining conversions and output include the pad characters as if they had been read from the input.
3. If the **bs=***expr* operand is specified and no conversion other than **sync** or **noerror** is requested, the resulting data will be written to the output as a single block, and the remaining steps are omitted.
4. If the **swab** conversion is specified, each pair of input data bytes will be swapped. If there is an odd number of bytes in the input block, the results are unspecified.
5. Any remaining conversions (**block**, **unblock**, **lcase** and **ucase**) will be performed. These conversions will operate on the input data independently of the input blocking; an input or output fixed-length record may span block boundaries.
6. The data resulting from input or conversion or both will be aggregated into output blocks of the specified size. After the end of input is reached, any remaining output will be written as a block without padding if **conv=sync** is not specified; thus the final output block may be shorter than the output block size.

OPTIONS

None.

OPERANDS

The following operands are supported:

if=*file* Specify the input pathname; the default is standard input.

of=*file* Specify the output pathname; the default is standard output. If the **seek=***expr* conversion is not also specified, the output file will be truncated before the copy begins, unless **conv=notrunc** is specified. If **seek=***expr* is specified, but **conv=notrunc** is not, the effect of the copy will be to preserve the blocks in the output file over which *dd* seeks, but no other portion of the output file will be preserved. (If the size of the seek plus the size of the input file is less than the previous size of the output file, the output

file will be shortened by the copy.)

ibs=*expr*
Specify the input block size, in bytes, by *expr* (default is 512).

obs=*expr*
Specify the output block size, in bytes, by *expr* (default is 512).

bs=*expr* Set both input and output block sizes to *expr* bytes, superseding **ibs=** and **obs=**. If no conversion other than **sync**, **noerror** and **notrunc** is specified, each input block will be copied to the output as a single block without aggregating short blocks.

cbs=*expr*
Specify the conversion block size for **block** and **unblock** in bytes by *expr* (default is zero). If **cbs=** is omitted or given a value of zero, using **block** or **unblock** produces unspecified results.

This option is used only if ASCII or EBCDIC conversion is specified. For the **ascii** operand, the input is handled as described for the **unblock** operand except that characters are converted to ASCII before the trailing spaces characters are deleted. For the **ebcdic** and **ibm** operands, the input is handled as described for the **block** operand except that the characters are converted to EBCDIC or IBM EBCDIC after the trailing spaces characters are added.

skip=*n* Skip *n* input blocks (using the specified input block size) before starting to copy. On seekable files, the implementation will read the blocks or seek past them; on non-seekable files, the blocks will be read and the data will be discarded.

seek=*n* Skip *n* blocks (using the specified output block size) from beginning of output file before copying. On non-seekable files, existing blocks will be read and space from the current end-of-file to the specified offset, if any, filled with null bytes; on seekable files, the implementation will seek to the specified offset or read the blocks as described for non-seekable files.

count=*n*
Copy only *n* input blocks.

conv=*value***[,***value* ...**]**
Where *value*s are comma-separated symbols from the following list.

EX **ascii** Convert EBCDIC to ASCII. See Table 3-4 on page 254.

EX **ebcdic** Convert ASCII to EBCDIC. See Table 3-4 on page 254.

EX **ibm** Convert ASCII to a different EBCDIC set. See Table 3-5 on page 255.

The **ascii**, **ebcdic** and **ibm** values are mutually exclusive.

block Treat the input as a sequence of newline-character-terminated or end-of-file-terminated variable-length records independent of the input block boundaries. Each record is converted to a record with a fixed length specified by the conversion block size. Any newline character is removed from the input line; space characters are appended to lines that are shorter than their conversion block size to fill the block. Lines that are longer than the conversion block size are truncated to the largest number of characters that will fit into that size; the number of truncated lines is reported (see **STDERR** below).

The **block** and **unblock** values are mutually exclusive.

unblock
: Convert fixed-length records to variable length. Read a number of bytes equal to the conversion block size (or the number of bytes remaining in the input, if less than the conversion block size), delete all trailing space characters, and append a newline character.

lcase
: Map upper-case characters specified by the LC_CTYPE keyword **tolower** to the corresponding lower-case character. Characters for which no mapping is specified will not be modified by this conversion.

  The **lcase** and **ucase** symbols are mutually exclusive.

ucase
: Map lower-case characters specified by the LC_CTYPE keyword **toupper** to the corresponding upper-case character. Characters for which no mapping is specified will not be modified by this conversion.

swab
: Swap every pair of input bytes. If the current input record is an odd number of bytes, the last byte in the input record is ignored.

noerror
: Do not stop processing on an input error. When an input error occurs, a diagnostic message will be written on standard error, followed by the current input and output block counts in the same format as used at completion (see **STDERR**). If the **sync** conversion is specified, the missing input will be replaced with null bytes and processed normally; otherwise, the input block will be omitted from the output.

notrunc
: Do not truncate the output file. Preserve blocks in the output file not explicitly written by this invocation of the *dd* utility. (See also the preceding **of=***file* operand.)

sync
: Pad every input block to the size of the **ibs=** buffer, appending null bytes. (If either **block** or **unblock** is also specified, append space characters, rather than null bytes.)

The behaviour is unspecified if operands other than **conv=** are specified more than once.

For the **bs=**, **cbs=**, **ibs=** and **obs=** operands, the application must supply an expression specifying a size in bytes. The expression, *expr*, can be:

1. a positive decimal number
2. a positive decimal number followed by k, specifying multiplication by 1024
3. a positive decimal number followed by b, specifying multiplication by 512
4. two or more positive decimal numbers (with or without k or b) separated by x, specifying the product of the indicated values.

All of the operands will be processed before any input is read.

EX The following two tables display the octal number character values used for the **ascii** and **ebcdic** conversions (first table) and for the **ibm** conversion (second table). In both tables, the ASCII values are the row and column headers and the EBCDIC values are found at their intersections. For example, ASCII 0012 (LF) is the second row, third column, yielding 0045 in EBCDIC. The inverted tables (for EBCDIC to ASCII conversion) are not shown, but are in one-to-one correspondence with these tables. The differences between the two tables are highlighted by small boxes drawn around five entries.

Table 3-4 ASCII to EBCDIC Conversion

	0	1	2	3	4	5	6	7
0000	0000 NUL	0001 SOH	0002 STX	0003 ETX	0067 EOT	0055 ENQ	0056 ACK	0057 BEL
0010	0026 BS	0005 HT	0045 LF	0013 VT	0014 FF	0015 CR	0016 SO	0017 SI
0020	0020 DLE	0021 DC1	0022 DC2	0023 DC3	0074 DC4	0075 NAK	0062 SYN	0046 ETB
0030	0030 CAN	0031 EM	0077 SUB	0047 ESC	0034 IFS	0035 IGS	0036 IRS	0037 ITB
0040	0100 Sp	0132 !	0177 "	0173 #	0133 $	0154 %	0120 &	0175 '
0050	0115 (	0135)	0134 *	0116 +	0153 ,	0140 -	0113 .	0141 /
0060	0360 0	0361 1	0362 2	0363 3	0364 4	0365 5	0366 6	0367 7
0070	0370 8	0371 9	0172 :	0136 ;	0114 <	0176 =	0156 >	0157 ?
0100	0174 @	0301 A	0302 B	0303 C	0304 D	0305 E	0306 F	0307 G
0110	0310 H	0311 I	0321 J	0322 K	0323 L	0324 M	0325 N	0326 O
0120	0327 P	0330 Q	0331 R	0342 S	0343 T	0344 U	0345 V	0346 W
0130	0347 X	0350 Y	0351 Z	0255 [	0340 \	0275]	0232	0155 _
0140	0171 `	0201 a	0202 b	0203 c	0204 d	0205 e	0206 f	0207 g
0150	0210 h	0211 i	0221 j	0222 k	0223]	0224 m	0225 n	0226 o
0160	0227 p	0230 q	0231 r	0242 s	0243 t	0244 u	0245 v	0246 w
0170	0247 x	0250 y	0251 z	0300 {	0117 |	0320 }	0137 ¬	0007 DEL
0200	0040 DS	0041 SOS	0042 FS	0043 WUS	0044 BYP	0025 NL	0006 RNL	0027 POC
0210	0050 SA	0051 SFE	0052 SM	0053 CSP	0054 MFA	0011 SPS	0012 RPT	0033 CU1
0220	0060	0061	0032 UBS	0063 IR	0064 PP	0065 TRN	0066 NBS	0010 GE
0230	0070 SBS	0071 IT	0072 RFF	0073 CU3	0004 SEL	0024 RES	0076	0341
0240	0101	0102	0103	0104	0105	0106	0107	0110
0250	0111	0121	0122	0123	0124	0125	0126	0127
0260	0130	0131	0142	0143	0144	0145	0146	0147
0270	0150	0151	0160	0161	0162	0163	0164	0165
0300	0166	0167	0170	0200	0212	0213	0214	0215
0310	0216	0217	0220	0152 ¦	0233	0234	0235	0236
0320	0237	0240	0252	0253	0254	0112 ¢	0256	0257
0330	0260	0261	0262	0263	0264	0265	0266	0267
0340	0270	0271	0272	0273	0274	0241	0276	0277
0350	0312	0313	0314 ⌋	0315	0316 ⅄	0317	0332	0333
0360	0334	0335	0336	0337	0352	0353	0354 ⊣	0355
0370	0356	0357	0372 ⏐	0373	0374	0375	0376	0377 EO

	0	1	2	3	4	5	6	7
0000	0000 NUL	0001 SOH	0002 STX	0003 ETX	0067 EOT	0055 ENQ	0056 ACK	0057 BEL
0010	0026 BS	0005 HT	0045 LF	0013 VT	0014 FF	0015 CR	0016 SO	0017 SI
0020	0020 DLE	0021 DC1	0022 DC2	0023 DC3	0074 DC4	0075 NAK	0062 SYN	0046 ETB
0030	0030 CAN	0031 EM	0077 SUB	0047 ESC	0034 IFS	0035 IGS	0036 IRS	0037 ITB
0040	0100 Sp	0132 !	0177 "	0173 #	0133 $	0154 %	0120 &	0175 '
0050	0115 (	0135)	0134 *	0116 +	0153 ,	0140 -	0113 .	0141 /
0060	0360 0	0361 1	0362 2	0363 3	0364 4	0365 5	0366 6	0367 7
0070	0370 8	0371 9	0172 :	0136 ;	0114 <	0176 =	0156 >	0157 ?
0100	0174 @	0301 A	0302 B	0303 C	0304 D	0305 E	0306 F	0307 G
0110	0310 H	0311 I	0321 J	0322 K	0323 L	0324 M	0325 N	0326 O
0120	0327 P	0330 Q	0331 R	0342 S	0343 T	0344 U	0345 V	0346 W
0130	0347 X	0350 Y	0351 Z	0255 [	0340 \	0275]	0137 ¬	0155 _
0140	0171 `	0201 a	0202 b	0203 c	0204 d	0205 e	0206 f	0207 g
0150	0210 h	0211 i	0221 j	0222 k	0223]	0224 m	0225 n	0226 o
0160	0227 p	0230 q	0231 r	0242 s	0243 t	0244 u	0245 v	0246 w
0170	0247 x	0250 y	0251 z	0300 {	0117 |	0320 }	0241	0007 DEL
0200	0040 DS	0041 SOS	0042 FS	0043 WUS	0044 BYP	0025 NL	0006 RNL	0027 POC
0210	0050 SA	0051 SFE	0052 SM	0053 CSP	0054 MFA	0011 SPS	0012 RPT	0033 CU1
0220	0060	0061	0032 UBS	0063 IR	0064 PP	0065 TRN	0066 NBS	0010 GE
0230	0070 SBS	0071 IT	0072 RFF	0073 CU3	0004 SEL	0024 RES	0076	0341
0240	0101	0102	0103	0104	0105	0106	0107	0110
0250	0111	0121	0122	0123	0124	0125	0126	0127
0260	0130	0131	0142	0143	0144	0145	0146	0147
0270	0150	0151	0160	0161	0162	0163	0164	0165
0300	0166	0167	0170	0200	0212	0213	0214	0215
0310	0216	0217	0220	0232	0233	0234	0235	0236
0320	0237	0240	0252	0253	0254	0255 [	0256	0257
0330	0260	0261	0262	0263	0264	0265	0266	0267
0340	0270	0271	0272	0273	0274	0275]	0276	0277
0350	0312	0313	0314 ⌋	0315	0316 ⅄	0317	0332	0333
0360	0334	0335	0336	0337	0352	0353	0354 ⊣	0355
0370	0356	0357	0372 ⏐	0373	0374	0375	0376	0377 EO

Table 3-5 ASCII to IBM EBCDIC Conversion

STDIN

If no **if=** operand is specified, the standard input will be used. See **INPUT FILES**.

INPUT FILES

The input file can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *dd*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the classification of characters as upper- or lower-case, and the mapping of characters from one case to the other.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

For SIGINT, the *dd* utility will write status information to standard error before exiting. It will take the standard action for all other signals; see **ASYNCHRONOUS EVENTS** in the **XBD** specification, **Section 2.1**, **Utility Description Defaults**.

STDOUT

If no **of=** operand is specified, the standard output will be used. The nature of the output depends on the operands selected.

STDERR

On completion, *dd* will write the number of input and output blocks to standard error. In the POSIX locale the following formats will be used:

```
"%u+%u records in\n", <number of whole input blocks>,
<number of partial input blocks>

"%u+%u records out\n", <number of whole output blocks>,
<number of partial output blocks>
```

A partial input block is one for which *read*( ) returned less than the input block size. A partial output block is one that was written with fewer bytes than specified by the output block size.

In addition, when there is at least one truncated block, the number of truncated blocks will be written to standard error. In the POSIX locale, the format is:

```
"%u truncated %s\n", <number of truncated blocks>, "record" (if
<number of truncated blocks> is one) "records" (otherwise)
```

Diagnostic messages may also be written to standard error.

OUTPUT FILES

If the **of=** operand is used, the output will be the same as described in **STDOUT**.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The input file was copied successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If an input error is detected and the **noerror** conversion has not been specified, any partial output block will be written to the output file, a diagnostic message will be written, and the copy operation will be discontinued. If some other error is detected, a diagnostic message will be written and the copy operation will be discontinued.

APPLICATION USAGE

The input and output block size can be specified to take advantage of raw physical I/O.

EXAMPLES

The following command:

```
dd if=/dev/rmt0h  of=/dev/rmt1h
```

copies from tape drive 0 to tape drive 1, using a common historical device naming convention.

The following command:

```
dd ibs=10  skip=1
```

strips the first 10 bytes from standard input.

This example reads an EBCDIC tape blocked ten 80-byte EBCDIC card images per block into the ASCII file **x**:

```
dd if=/dev/tape of=x ibs=800 cbs=80 conv=ascii,lcase
```

FUTURE DIRECTIONS

None.

SEE ALSO

sed, *tr*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO/IEC 9945-2: 1993 standard.

NAME

delta – make a delta (change) to an SCCS file (**DEVELOPMENT**)

SYNOPSIS

EX `deta [-nps][-g list][-m mrlist][-r SID][-y[comment]] file...`

DESCRIPTION

The *delta* utility is used to permanently introduce into the named SCCS files changes that were made to the files retrieved by *get* (called the *g-files*, or generated files).

OPTIONS

The *delta* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the **–y** option has an optional option-argument. This optional option-argument cannot be presented as a separate argument. The following options are supported:

–r *SID* Uniquely identify which delta is to be made to the SCCS file. The use of this option is necessary only if two or more outstanding *get* commands for editing (*get* **–e**) on the same SCCS file were done by the same person (login name). The SID value specified with the **–r** option can be either the SID specified on the *get* command line or the SID to be made as reported by the *get* utility; see *get*.

–s Suppress the report to standard output of the activity associated with each *file*. See **STDOUT**.

–n Specify retention of the edited *g-file* (normally removed at completion of delta processing).

–g *list* Specify a *list*, (see *get* for the definition of *list*) of deltas that are to be ignored when the file is accessed at the change level (SID) created by this delta.

–m *mrlist*

Specify a modification request (MR) number that must be supplied as the reason for creating the new delta. This is used if the SCCS file has the **v** flag set; see *admin*.

If **–m** is not used and the standard input is a terminal, the prompt described in **STDOUT** is written to standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued.

MRs in a list are separated by blanks. An unescaped newline 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 the MR number validation program, *delta* terminates. (It is assumed that the MR numbers were not all valid.)

–y[*comment***]**

Describe the reason for making the delta. This is an arbitrary group of lines that would meet the definition of a text file. Systems support *comment*s from zero to 512 bytes and may support longer values. A null string (specified as either **–y**, **–y**" " or in response to a prompt for a comment) is considered a valid *comment*.

If **–y** is not specified and the standard input is a terminal, the prompt described in **STDOUT** is written to standard output before the standard input is read; if the standard input is not a terminal, no prompt is issued. An unescaped newline character terminates the comment text.

The **–y** option is required if the *file* operand is specified as –.

–p Write (to standard output) the SCCS file differences before and after the delta is applied in *diff* format; see *diff*.

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *delta* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only in the following cases:

- A prompt is issued for the **–m** or **–y** options.
- The *file* operand is specified as –.

INPUT FILES

Input files are text files whose data is to be included in the SCCS files. If the first character of any line of an input file is SOH (binary 001), the results are unspecified.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *delta*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is used only for the following messages in the POSIX locale:

- Prompts (see the –**m** and –**y** options) in the following formats:

  ```
  "MRs? "
  ```

  ```
  "comments? "
  ```

  The MR prompt, if written, always precedes the comments prompt.

- A report of each *file*'s activities (unless the –**s** option is specified) in the following format:

  ```
  "%d inserted\n%d deleted\n%d unchanged\n",
  <number of lines inserted>, <number of lines deleted>,
  <number of lines unchanged>
  ```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Any SCCS files updated are files of an unspecified format.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

A version of *delta* that fully supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** may be introduced in a future issue.

SEE ALSO

admin, *diff*, *get*, *prs*, *rmdel*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

NAME

df – report free disk space

SYNOPSIS

EX `df [-k][ -P|-t][`*`file`*`...]`

DESCRIPTION

EX The *df* utility writes the amount of available space and file slots for file systems on which the invoking user has appropriate read access. File systems are specified by the *file* operands; when none are specified, information is written for all file systems. The format of the default output from *df* is unspecified, but all space figures will be reported in 512-byte units, unless the **–k** option is specified. This output contains at least the file system names, amount of available
EX space on each of these file systems, and the number of free file slots, or *inodes*, available; when **–t** is specified, the output contains the total allocated space as well.

OPTIONS

The *df* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–k Use 1024-byte units, instead of the default 512-byte units, when writing space figures.

–P Produce output in the format described in **STDOUT**.

EX **–t** Include total allocated-space figures in the output.

OPERANDS

The following operand is supported:

file A pathname of a file within the hierarchy of the desired file system. If a file other than a
EX FIFO, a regular file, a directory or a special file representing the device containing the file system (for example, **/dev/dsk/0s1**) is specified, the results are unspecified. Otherwise, *df* will write the amount of free space in the file system containing the specified *file* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *df*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When both the –**k** and –**P** options are specified, the following header line will be written (in the POSIX locale):

```
"Filesystem 1024-blocks Used Available Capacity Mounted on\n"
```

When the –**P** option is specified without the –**k** option, the following header line will be written (in the POSIX locale):

```
"Filesystem 512-blocks Used Available Capacity Mounted on\n"
```

The implementation may adjust the spacing of the header line and the individual data lines so that the information is presented in orderly columns.

The remaining output with –**P** will consist of one line of information for each specified file system. These lines are formatted as follows:

```
"%s %d %d %d %d%% %s\n", <file system name>, <total space>,
<space used>, <space free>, <percentage used>, <file system root>
```

In the following list, all quantities expressed in 512-byte units (1024-byte when –**k** is specified) will be rounded up to the next higher unit. The fields are:

<file system name>
The name of the file system, in an implementation-dependent format.

<total space>
The total size of the file system in 512-byte units. The exact meaning of this figure is implementation-dependent, but should include *<space used>*, *<space free>*, plus any space reserved by the system not normally available to a user.

<space used>
The total amount of space allocated to existing files in the file system, in 512-byte units.

<space free>
The total amount of space available within the file system for the creation of new files by unprivileged users, in 512-byte units. When this figure is less than or equal to zero, it is not possible to create any new files on the file system without first deleting others, unless the process has appropriate privileges. The figure written may be less than zero.

<percentage used>
The percentage of the normally available space that is currently allocated to all files on the file system. This is calculated using the fraction:

```
<space used>/(<space used>+<space free>)
```

expressed as a percentage. This percentage may be greater than 100 if *<space free>* is less than zero. The percentage value is expressed as a positive integer, with any fractional result causing it to be rounded to the next highest integer.

<file system root>
The directory below which the file system hierarchy appears.

EX The output format is unspecified when **–t** is used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
On most systems, the ‘‘name of the file system, in an implementation-dependent format’’ will be the special file on which the file system is mounted.

EXAMPLES

1. The following example writes portable information about the **/usr** file system:

   ```
   df -P /usr
   ```

2. Assuming that **/usr/src** is part of the **/usr** file system, the following will do the same as the previous example:

   ```
   df -P /usr/src
   ```

FUTURE DIRECTIONS
None.

SEE ALSO
find.

CHANGE HISTORY
First released in Issue 2.

Issue 3
Functionally equivalent to the entry in Issue 2.

Issue 4
Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

diff – compare two files

SYNOPSIS

EX `diff [-c| -e| -f| -C n][-br] file1 file2`

DESCRIPTION

The *diff* utility will compare the contents of *file1* and *file2* and write to standard output a list of changes necessary to convert *file1* into *file2*. This list should be minimal. No output will be produced if the files are identical.

OPTIONS

The *diff* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–b Cause any amount of white space at the end of a line to be treated as a single newline character (that is, the white-space characters preceding the newline character are ignored) and other strings of white-space characters, not including newline characters, to compare equal.

–c Produce output in a form that provides three lines of context.

–C *n* Produce output in a form that provides *n* lines of context (where *n* will be interpreted as a positive decimal integer).

–e Produce output in a form suitable as input for the *ed* utility, which can then be used to convert *file1* into *file2*.

EX **–f** Produce output in an alternative form, similar in format to **–e**, but unsuitable as input for the *ed* utility, and in the opposite order.

–r Apply *diff* recursively to files and directories of the same name when *file1* and *file2* are both directories.

OPERANDS

The following operands are supported:

file1
file2 A pathname of a file be compared. If either the *file1* or *file2* operand is –, the standard input will be used in its place.

If both *file1* and *file2* are directories, *diff* will not compare block special files, character special files or FIFO special files to any files and will not compare regular files to directories. The system documentation will specify the behaviour of *diff* on implementation-specific file types not specified by the **XSH** specification when found in directories. Further details are as specified in **Diff Directory Comparison Format** on page 265.

If only one of *file1* and *file2* is a directory, *diff* will be applied to the non-directory file and the file contained in the directory file with a filename that is the same as the last component of the non-directory file.

STDIN

The standard input will be used only if one of the *file1* or *file2* operands references standard input. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *diff*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

LC_TIME
Determine the locale for affecting the format of file timestamps written with the **–C** and **–c** options.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the locale for affecting the timezone used for calculating file timestamps written with the **–C** and **–c** options.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Diff Directory Comparison Format

If both *file1* and *file2* are directories, the following output formats will be used.

In the POSIX locale, each file that is present in only one directory will be reported using the following format:

```
"Only in %s: %s\n", <directory pathname>, <filename>
```

In the POSIX locale, subdirectories that are common to the two directories may be reported with the following format:

```
"Common subdirectories: %s and %s\n", <directory1 pathname>,
<directory2 pathname>
```

For each file common to the two directories if the two files are not to be compared, the following format shall be used in the POSIX locale:

```
"File %s is a %s while file %s is a %s\n",
<directory1 pathname>, <file type of directory1 pathname>,
<directory2 pathname>, <file type of directory2 pathname>
```

For each file common to the two directories, if the files are to be compared and are identical, no output shall be written. If the two files differ, the following format shall be written:

"diff %s %s %s\n", <*diff_options*>, <*filename1*>, <*filename2*>

where <*diff_options*> are the options as specified on the command line. Depending on these options, one of the following output formats will be used to write the differences.

All directory pathnames listed in this section will be relative to the original command line arguments. All other names of files listed in this section will be filenames (pathname components).

Diff Default Output Format

EX The default (without **–e**, **–f**, **–c** or **–C** options) *diff* utility output contains lines of these forms:

"%da%d\n", <*num1*>, <*num2*>

"%da%d,%d\n", <*num1*>, <*num2*>, <*num3*>

"%dd%d\n", <*num1*>, <*num2*>

"%d,%dd%d\n", <*num1*>, <*num2*>, <*num3*>

"%dc%d\n", <*num1*>, <*num2*>

"%d,%dc%d\n", <*num1*>, <*num2*>, <*num3*>

"%dc%d,%d\n", <*num1*>, <*num2*>, <*num3*>

"%d,%dc%d,%d\n", <*num1*>, <*num2*>, <*num3*>, <*num4*>

These lines resemble *ed* subcommands to convert *file1* into *file2*. The line numbers before the action letters pertain to *file1*; those after pertain to *file2*. Thus, by exchanging a for d and reading the line in reverse order, one can also determine how to convert *file2* into *file1*. As in *ed*, identical pairs (where *num1* = *num2*) are abbreviated as a single number.

Following each of these lines, *diff* will write to standard output all lines affected in the first file using the format:

"<Δ%s", <*line*>

and all lines affected in the second file using the format:

">Δ%s", <*line*>

If there are lines affected in both *file1* and *file2* (as with the **c** subcommand), the changes are separated with a line consisting of three hyphens:

"---\n"

Diff –e Output Format

With the **–e** option, a script will be produced that will, when provided as input to *ed*, along with an appended **w** (write) command, convert *file1* into *file2*. Only the **a** (append), **c** (change), **d** (delete), **i** (insert), and **s** (substitute) commands of *ed* will be used in this script. Text lines, except those consisting of the single character period (.), will be output as they appear in the file.

Diff –f Output Format

EX With the **–f** option, an alternative format of script will be produced. It will be similar to that produced by **–e**, with the following differences:

1. It will be expressed in reverse sequence; the output of **–e** will order changes from the end of the file to the beginning; the **–f** from beginning to end.
2. The command form <*lines*> <*command-letter*> used by **–e** will be reversed. For example, 10c with **–e** would be c10 with **–f**.
3. The form used for ranges of line numbers will be space-character-separated, rather than comma-separated.

Diff –c or –C Output Format

With the **–c** or **–C** option, the output format will consist of affected lines along with surrounding lines of context. The affected lines will show which ones need to be deleted or changed in *file1*, and those added from *file2*. With the **–c** option, three lines of context, if available, will be written before and after the affected lines. With the **–C** option, the user can specify how many lines of context will be written. The exact format follows.

The name and last modification time of each file will be output in the following format:

```
"*** %s %s\n", file1, <file1 time stamp> "--- %s %s\n", file2,
<file2 time stamp>
```

and a string of 15 asterisks:

```
"***************\n"
```

Each <*file*> field will be the pathname of the corresponding file being compared. The pathname written for standard input is unspecified.

In the POSIX locale, each <*timestamp*> field will be equivalent to the output from the following command:

```
date "+%a %b %e %T %Y"
```

without the trailing newline character, executed at the time of last modification of the corresponding file (or the current time, if the file is standard input).

Then, the following output formats will be applied for every set of changes.

First, the range of lines in *file1* will be written in the following format:

```
"*** %d,%d ****\n", <beginning line number>, <ending line number>
```

Next, the affected lines along with lines of context (unaffected lines) will be written. Unaffected lines will be written in the following format:

```
"␣␣%s", <unaffected_line>
```

Deleted lines will be written as:

```
"-␣%s", <deleted_line>
```

Changed lines will be written as:

```
"!␣%s", <changed_line>
```

Next, the range of lines in *file2* will be written in the following format:

```
"--- %d,%d ----\n", <beginning line number>, <ending line number>
```

Then, lines of context and changed lines will be written as described in the previous formats. Lines added from *file2* will be written in the following format:

```
"+␣%s", <added_line>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 No differences were found.
1 Differences were found.
>1 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If lines at the end of a file are changed and other lines are added, *diff* output may show this as a delete and add, as a change, or as a change and add; *diff* is not expected to know which happened and users should not care about the difference in output as long as it clearly shows the differences between the files.

EXAMPLES

If **dir1** is a directory containing a directory named **x**, **dir2** is a directory containing a directory named **x**, **dir1/x** and **dir2/x** both contain files named **date.out**, and **dir2/x** contains a file named **y**, the command:

```
diff -r dir1 dir2
```

could produce output similar to:

```
Common subdirectories: dir1/x and dir2/x
Only in dir2/x: y
diff -r dir1/x/date.out dir2/x/date.out
1c1
< Mon Jul  2 13:12:16 PDT 1990
---
> Tue Jun 19 21:41:39 PDT 1990
```

FUTURE DIRECTIONS

None.

SEE ALSO

cmp, *comm*, *dircmp*, *ed*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

dircmp – directory comparison (**TO BE WITHDRAWN**)

SYNOPSIS

EX `dircmp [-ds] dir1 dir2`

DESCRIPTION

The *dircmp* utility examines the directory hierarchies specified by *dir1* and *dir2* and generates various tabulated information about the contents of the directories. Sorted listings of files that are unique to each directory are generated for all the options. If no option is specified, a list is output that indicates whether the filenames common to both directories have the same contents.

OPTIONS

The *dircmp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–d Compare the contents of files with the same name in both directories and output a list indicating what must be changed in the two files to bring them into agreement. The list format is described in *diff*.

–s Suppress messages about identical files.

OPERANDS

The following operands are supported:

dir1
dir2 A pathname of a directory to be compared.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *dircmp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the ordering of the output.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output format of *dircmp* is unspecified, but includes:

- lists of pathnames in either *dir1* or *dir2*, but not both
- when **–d** is given, listings of differences between files, as produced by *diff*.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred. (Differences in directory contents are not considered errors.)

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

The *dircmp* utility will be withdrawn from a future issue. The *diff* **–R** command should be used instead.

SEE ALSO

cmp, *diff*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

dirname – return directory portion of pathname

SYNOPSIS

`dirname` *`string`*

DESCRIPTION

The *string* operand will be treated as a pathname, as defined in **pathname** (see the **XBD** specification, **Chapter 2**, **Glossary**). The string *string* will be converted to the name of the directory containing the filename corresponding to the last pathname component in *string*, performing actions equivalent to the following steps in order:

1. If *string* is **//**, skip steps 2 to 5.
2. If *string* consists entirely of slash characters, *string* will be set to a single slash character. In this case, skip steps 3 to 8.
3. If there are any trailing slash characters in *string*, they will be removed.
4. If there are no slash characters remaining in *string*, *string* will be set to a single period character. In this case, skip steps 5 to 8.
5. If there are any trailing non-slash characters in *string*, they will be removed.
6. If the remaining *string* is **//**, it is implementation-dependent whether steps 7 and 8 are skipped or processed.
7. If there are any trailing slash characters in *string*, they will be removed.
8. If the remaining *string* is empty, *string* will be set to a single slash character.

The resulting string will be written to standard output.

OPTIONS

None.

OPERANDS

The following operand is supported:

string A string.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *dirname*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *dirname* utility will write a line to the standard output in the following format:

```
"%s\n", <resulting string>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The definition of *pathname* specifies implementation-dependent behaviour for pathnames starting with two slash characters. Therefore, applications must not arbitrarily add slashes to the beginning of a pathname unless they can ensure that there are more or less than two or are prepared to deal with the implementation-dependent consequences.

EXAMPLES

Command	Results
`dirname /`	`/`
`dirname //`	`/` or `//`
`dirname /a/b/`	`/a`
`dirname //a//b//`	`//a`
`dirname`	*unspecified*
`dirname a`	`. ($? = 0)`
`dirname ""`	`. ($? = 0)`
`dirname /a`	`/`
`dirname /a/b`	`/a`
`dirname a/b`	`a`

FUTURE DIRECTIONS
None.

SEE ALSO
basename, Section 2.5 on page 27.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

dis – disassembler (**DEVELOPMENT**) (**OPTIONAL**) (**TO BE WITHDRAWN**)

SYNOPSIS

PI `dis [-olV][-F function]... [-l string] file...`

DESCRIPTION

The *dis* utility 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.

OPTIONS

The *dis* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–o Write numbers in octal. The default is hexadecimal.

–L Invoke a lookup of C-language source labels in the symbol table for subsequent writing to standard output.

–V Write the version number of the disassembler to standard error.

–F *function*
Disassemble only the named *function* in each object file specified on the command line.

–l *string*
Disassemble the library file specified as *string*. For example, the command *dis* **–l m** will disassemble the math library.

OPERANDS

The following operands are supported:

file A pathname of an object file or an archive (see *ar*) of object files.

STDIN

Not used.

INPUT FILES

The input files are object files or archives of object files, or both.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *dis*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output consists of an assembly listing of unspecified format.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
The *dircmp* utility will be withdrawn from a future issue.

SEE ALSO
ar, *cc*, *c89*.

CHANGE HISTORY
First released in Issue 2.

Issue 3
Functionally equivalent to the entry in Issue 2.

Issue 4
Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

du – estimate file space usage

SYNOPSIS

EX OB

```
du [-a| -s][-kx][-r][file ...]
```

DESCRIPTION

By default, the *du* utility writes to standard output the size of the file space allocated to, and the size of the file space allocated to each subdirectory of, the file hierarchy rooted in each of the specified files. The size of the file space allocated to a file of type directory is defined as the sum total of space allocated to all files in the file hierarchy rooted in the directory plus the space allocated to the directory itself.

When *du* cannot *stat*( ) files or *stat*( ) or read directories, it will report an error condition and the final exit status will be affected. Files with multiple links will be counted and written for only one entry. The directory entry that is selected in the report is unspecified. By default, file sizes are written in 512-byte units, rounded up to the next 512-byte unit.

OPTIONS

The *du* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a In addition to the default output, report the size of each file not of type directory in the file hierarchy rooted in the specified file. Regardless of the presence of the **–a** option, non-directories given as *file* operands will always be listed.

–k Write the files sizes in units of 1024 bytes, rather than the default 512-byte units.

EX OB **–r** Generate messages about directories that cannot be read, files that cannot be opened, and so forth. This is the default case.

–s Instead of the default output, report only the total sum for each of the specified files.

–x When evaluating file sizes, evaluate only those files that have the same device as the file specified by the *file* operand.

OPERANDS

The following operand is supported:

file The pathname of a file whose size is to be written. If no *file* is specified, the current directory is used.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *du*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The output from *du* consists of the amount of the space allocated to a file and the name of the file, in the following format:

```
"%d %s\n", <size>, <pathname>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Historical file systems provided no way to obtain exact figures for the space allocation given to files. There are two known areas of inaccuracies in traditional file systems: cases of *indirect blocks* being used by the file system or *sparse* files yielding incorrectly high values. An indirect block is space used by the file system in the storage of the file, but that need not be counted in the space allocated to the file. A *sparse* file is one in which an *lseek*( ) call has been made to a position beyond the end of the file and data has subsequently been written at that point. A file system need not allocate all the intervening zero-filled blocks to such a file. It is up to the implementation to define exactly how accurate its methods are.

EXAMPLES
None.

FUTURE DIRECTIONS
The obsolescent **–r** option may be withdrawn from a future issue.

SEE ALSO
ls.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

echo – write arguments to standard output

SYNOPSIS

echo **[***string* **...]**

DESCRIPTION

The *echo* utility will write its arguments to standard output, followed by a newline character. If there are no arguments, only the newline character will be written.

OPTIONS

The *echo* utility will not recognise the –– argument in the manner specified by Guideline 10 of the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**; –– will be recognised as a string operand.

Implementations need not support any options.

OPERANDS

The following operands are supported:

EX *string* A string to be written to standard output. If any operand is "–n", it will be treated as a string, not an option. The following character sequences will be recognised within any of the arguments:

\a	Write an alert character.
\b	Write a backspace character.
\c	Suppress the newline character that otherwise follows the final argument in the output. All characters following the \c in the arguments will be ignored.
\f	Write a form-feed character.
\n	Write a newline character.
\r	Write a carriage-return character.
\t	Write a tab character.
\v	Write a vertical-tab character.
****	Write a backslash character.
\0*num*	Write an 8-bit value that is the zero-, one-, two- or three-digit octal number *num*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *echo*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

EX *LC_CTYPE*
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *echo* utility arguments will be separated by single space characters and a newline character
EX will follow the last argument. Output transformations will occur based on the escape sequences in the input; see **OPERANDS**.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
It is not possible to use *echo* portably across all systems that are not XSI-conformant unless both **–n** (as the first argument) and escape sequences are omitted.

The *printf* utility can be used portably to emulate any of the traditional behaviours of the *echo* utility as follows:

- The XSI *echo* is equivalent to:

```
printf "%b\n" "$*"
```

- The BSD *echo* is equivalent to:

```
if [ "X$1" = "X-n" ]
then
       shift
       printf "%s" "$*"
else
       printf "%s\n" "$*"
fi
```

New applications are encouraged to use *printf* instead of *echo*.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

printf.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

ed – edit text

SYNOPSIS

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

OB

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

DESCRIPTION

The *ed* utility is a line-oriented text editor that uses two modes: *command mode* and *input mode*. In command mode the input characters are interpreted as commands, and in input mode they are interpreted as text. See **EXTENDED DESCRIPTION**.

OPTIONS

OB The *ed* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except for its non-standard usage of –.

The following options are supported:

–p *string*
Use *string* as the prompt string when in command mode. By default, there is no prompt string.

–s

OB – Suppress the writing of byte counts by **e**, **E**, **r** and **w** commands and of the ! prompt after a !*command*.

OPERANDS

The following operand is supported:

file If the *file* argument is given, *ed* will simulate an **e** command on the file named by the pathname, *file*, before accepting commands from the standard input.

STDIN

The standard input must be a text file consisting of commands, as described in **EXTENDED DESCRIPTION**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ed*:

HOME Determine the pathname of the user's home directory.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within regular expressions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

The *ed* utility will take the standard action for all signals (see **ASYNCHRONOUS EVENTS** in the **XBD** specification, **Section 2.1**, **Utility Description Defaults**) with the following exceptions:

SIGINT The *ed* utility will interrupt its current activity, write the string ?\n to standard output, and return to command mode (see **EXTENDED DESCRIPTION**).

SIGHUP
If the buffer is not empty and has changed since the last write, the *ed* utility will attempt to write a copy of the buffer in a file. First, the file named **ed.hup** in the current directory will be used; if that fails, the file named **ed.hup** in the directory named by the *HOME* environment variable will be used. In any case, the *ed* utility will exit without returning to command mode.

STDOUT

Various editing commands and the prompting feature (see **–p**) write to standard output, as described in **EXTENDED DESCRIPTION**.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files are text files whose formats are dependent on the editing commands given.

EXTENDED DESCRIPTION

The *ed* utility operates on a copy of the file it is editing; changes made to the copy will have no effect on the file until a **w** (write) command is given. The copy of the text is called the *buffer*.

Commands to *ed* have a simple and regular structure: zero, one or two *addresses* followed by a single-character *command*, possibly followed by parameters to that command. These addresses specify one or more lines in the buffer. Every command that requires addresses has default addresses, so that the addresses very often can be omitted. If the **–p** option is specified, the prompt string will be written to standard output before each command is read.

In general, only one command can appear on a line. Certain commands allow text to be input. 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 recognised; all input is merely collected. Input mode is terminated by entering a line consisting of two characters: a period (.) followed by a newline character. This line is not considered part of the input text.

Regular Expressions in ed

The *ed* utility supports basic regular expressions, as described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**. Since regular expressions in *ed* are always matched against single lines, never against any larger section of text, there is no way for a regular expression to match a newline character. A null RE is equivalent to the last RE encountered.

Regular expressions are used in addresses to specify lines, and in some commands (for example, the **s** substitute command) to specify portions of a line to be substituted.

Addresses in ed

Addressing in *ed* relates to the *current line*. Generally, the current line is the last line affected by a command. The *current line number* is the address (line number) of the current line. The exact effect on the current line number is discussed under the description of each command. The f, h, H, k, P, w, = and ! commands do not modify the current line number.

Addresses are constructed as follows:

1. The character . (period) addresses the current line.
2. The character $ addresses the last line of the buffer.
3. A positive decimal number *n* addresses the *n*th line of the buffer. The first line in the buffer is line number 1.
4. **mark name character** *x*, which must be a lower-case letter from the portable character set. Lines can be marked with the k command.
5. An RE enclosed by slashes (/) addresses the first line found by searching forward from the line following the current line toward the end of the buffer and stopping at the first line containing a string matching the RE. An address consisting of a null RE delimited by slashes (//) addresses the next line containing the last RE encountered. If necessary, the search will wrap around to the beginning of the buffer and continue up to and including the current line, so that the entire buffer is searched. Within the RE, the sequence \/ represents a literal slash instead of the RE delimiter.
6. An RE enclosed in question-marks (?) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. If necessary, the search wraps around to the end of the buffer and continues up to and including the current line. Within the RE, the sequence \? represents a literal question-mark instead of the RE delimiter.
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 can be omitted.
8. If an address begins with + or –, the addition or subtraction is taken with respect to the current line number; for example, –5 is understood to mean .–5.
9. If an address ends with + or –, then 1 will be 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 number less 2.
10. A comma (,) stands for the address pair 1,$, while a semicolon (;) stands for the pair .,$.

Commands 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 no addresses are given. If one address is given to a command that allows two addresses, the command will operate as if it were specified as:

```
given_address;.  command
```

If more addresses are given than such a command requires, the results are undefined.

Typically, addresses are separated from each other by a comma. They can also be separated by a semicolon. In the latter case, the current line number (.) is set to the first address, and only then will the second address be 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 corresponds to a line that does not precede, in the buffer, the line corresponding to the first address.

Commands in ed

In the following list of *ed* commands, the default addresses are shown in parentheses. The number of addresses shown in the default are the number expected by the command. The parentheses are not part of the address; they show that the given addresses are the default.

It is generally invalid for more than one command to appear on a line. However, any command (except **e**, **E**, **f**, **q**, **Q**, **r**, **w** and !) can be suffixed by the letter **l**, **n** or **p**; in which case, except for the **l**, **n** and **p** commands, the command will be executed and then the new current line will be written as described below under the **l**, **n** and **p** commands. When an **l**, **n** or **p** suffix is used with an **l**, **n** or **p** command, the command will write to standard output as described below, but it is unspecified whether the suffix writes the current line again in the requested format or whether the suffix has no effect. For example, the **pl** command (base **p** command with an **l** suffix) will either write just the current line or will write it twice once as specified for **p** and once as specified for **l**. Also, the **g**, **G**, **v** and **V** commands takes a command as a parameter.

Each address component can be preceded by zero or more blank characters. The command letter can be preceded by zero or more blank characters. If a suffix letter (**l**, **n** or **p**) is given, it must immediately follow the command.

The **e**, **E**, **f**, **r** and **w** commands take an optional *file* parameter, separated from the command letter by one or more blank characters.

If changes have been made in the buffer since the last **w** command that wrote the entire buffer, *ed* will warn the user if an attempt is made to destroy the editor buffer via the **e** or **q** commands. The *ed* utility will write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the **H** command) to standard output and will continue in command mode with the current line number unchanged. If the **e** or **q** command is repeated with no intervening command, it will take effect.

If an end-of-file is detected on standard input when a command is expected, the *ed* utility acts as if a **q** command had been entered.

If the closing delimiter of an RE or of a replacement string (for example, */*) in a **g**, **G**, **s**, **v** or **V** command would be the last character before a newline character, that delimiter can be omitted, in which case the addressed line is written. For example, the following pairs of commands are equivalent:

```
s/s1/s2 s/s1/s2/p
g/s1   g/s1/p
?s1    ?s1?
```

If an invalid command is entered, *ed* will write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the **H** command) to standard output and will continue in command mode with the current line number unchanged.

Append Command

Synopsis: `(.)a`
`<text>`
`.`

The **a** command reads the given text and appends it after the addressed line; the current line number will become the address of the last inserted line or, if there were none, the addressed line. Address 0 is valid for this command; it causes the appended text to be placed at the beginning of the buffer.

Change Command

Synopsis: `(.,.)c`
`<text>`
`.`

The **c** command deletes the addressed lines, then accepts input text that replaces these lines; the current line will be set to the address of the last line input; or, if there were none, at the line after the last line deleted; if the lines deleted were originally at the end of the buffer, the current line number will be set to the address of the new last line; if no lines remain in the buffer, the current line number will be set to zero.

Delete Command

Synopsis: `(.,.)d`

The **d** command deletes the addressed lines from the buffer. The address of the line after the last line deleted will become the current line number; if the lines deleted were originally at the end of the buffer, the current line number will be set to the address of the new last line; if no lines remain in the buffer, the current line number will be set to zero.

Edit Command

Synopsis: `e [`*`file`*`]`

The **e** command deletes the entire contents of the buffer and then reads in the file named by the pathname *file*. The current line number will be set to the address of the last line of the buffer. If no pathname is given, the currently remembered pathname, if any, will be used (see the **f** command). The number of bytes read will be written to standard output, unless the **–s** option was specified, in the following format:

```
"%d\n", <number of bytes read>
```

The name *file* will be remembered for possible use as a default pathname in subsequent **e**, **E**, **r** and **w** commands. If *file* is replaced by **!**, the rest of the line will be taken to be a shell command

line whose output is to be read. Such a shell command line is be remembered as the current *file*. All marks will be discarded upon the completion of a successful **e** command. If the buffer has changed since the last time the entire buffer was written, the user will be warned, as described previously.

Edit Without Checking Command

Synopsis: `E [`*`file`*`]`

The **E** command possesses all properties and restrictions of the **e** command except that the editor will not check to see if any changes have been made to the buffer since the last w command.

Filename Command

Synopsis: `f [`*`file`*`]`

If *file* is given, the **f** command will change the currently remembered pathname to *file*; whether the name is changed or not, it then will write the (possibly new) currently remembered pathname to the standard output in the following format:

`"%s\n", <`*`pathname`*`>`

The current line number is unchanged.

Global Command

Synopsis: `(1,$)g/`*`RE/command list`*

In the **g** command, the first step is to mark every line that matches the given *RE*. Then, for every such line, the given *command list* will be executed with the current line number set to the address of that line. When the **g** command completes, the current line number will have the value assigned by the last command in the command list. If there were no matching lines, the current line number will not be changed. A single command or the first of a list of commands will appear on the same line as the global command. All lines of a multi-line list except the last line will be ended with a backslash; the **a**, **i** and **c** commands and associated input are permitted. The . terminating input mode can be omitted if it would be the last line of the *command list*. An empty *command list* is equivalent to the **p** command. The use of the **g**, **G**, **v**, **V** and **!** commands in the *command list* produces undefined results. Any character other than space or newline can be used instead of a slash to delimit the *RE*. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash.

Interactive Global Command

Synopsis: `(1,$)G/`*`RE/`*

In the **G** command, the first step is to mark every line that matches the given *RE*. Then, for every such line, that line will be written, the current line number will be set to the address of that line, and any one command (other than one of the **a**, **c**, **i**, **g**, **G**, **v** and **V** commands) can be input and will be executed. A newline character acts as a null command (causing no action to be taken on the current line); an **&** causes the reexecution of the most recent non-null command executed within the current invocation of **G**. Note that the commands input as part of the execution of the **G** command can address and affect any lines in the buffer. The final value of the current line number will be the value set by the last command successfully executed. (Note that the last command successfully executed will be the **G** command itself if a command fails or the null command is specified.) If there were no matching lines, the current line number will not be changed. The **G** command can be terminated by a SIGINT signal. Any character other than

space or newline can be used instead of a slash to delimit the *RE* and the replacement. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash.

Help Command

Synopsis: `h`

The **h** command writes a short message to standard output that explains the reason for the most recent **?** notification. The current line number is unchanged.

Help-mode Command

Synopsis: `H`

The **H** command causes *ed* to enter a mode in which help messages (see the **h** command) will be written to standard output for all subsequent **?** notifications. The **H** command alternatively will turn this mode on and off; it is initially off. If the help-mode is being turned on, the **H** command also will explain the previous **?** notification, if there was one. The current line number is unchanged.

Insert Command

Synopsis:
```
(.)i
<text>
.
```

The **i** command inserts the given text before the addressed line; . will be left at the last inserted line or, if there was none, at the addressed line. This command differs from the **a** command only in the placement of the input text. Address 0 is invalid for this command.

Join Command

Synopsis: `(.,.+1)j`

The **j** command joins contiguous lines by removing the appropriate newline characters. If exactly one address is given, this command will do nothing. If lines are joined, the current line number will be set to the address of the joined line; otherwise, the current line number is unchanged.

Mark Command

Synopsis: `(.)k`*x*

The **m** command marks the addressed line with name *x*, which must be a lower-case letter from the portable character set. The address *'x* then refers to this line; the current line number is unchanged.

List Command

Synopsis: `(.,.)l`

The **l** command writes to standard output the addressed lines in a visually unambiguous form. The characters listed in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \r, \t, \v) will be written as the corresponding escape sequence; the \n in that table is not applicable. Non-printable characters not in the table will be written as one three-digit octal number (with a preceding backslash character) for each byte in the character (most significant byte first). If the size of a byte on the system is greater than nine bits, the format used for non-printable characters is implementation-dependent.

Long lines will be folded, with the point of folding indicated by writing backslash/newline character; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line will be marked with a $. An **l** command can be appended to any other command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w** or **!**. The current line number will be set to the address of the last line written.

Move Command

Synopsis: `(.,.)m`*address*

The **m** command repositions the addressed lines after the line addressed by *address*. Address 0 is valid for *address* and causes the addressed lines to be moved to the beginning of the buffer. It is an error if address *address* falls within the range of moved lines. The current line number will be set to the address of the last line moved.

Number Command

Synopsis: `(.,.)n`

The **n** command writes to standard output the addressed lines, preceding each line by its line number and a tab character; the current line number will be set to the address of the last line written. The **n** command can be appended to any command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w** or **!**.

Print Command

Synopsis: `(.,.)p`

The **p** command writes to standard output the addressed lines; the current line number will be set to the address of the last line written. The **p** command can be appended to any command other than **e**, **E**, **f**, **q**, **Q**, **r**, **w** or **!**.

Prompt Command

Synopsis: `P`

The **P** command causes *ed* to prompt with an asterisk (*) (or *string*, if **–p** is specified) for all subsequent commands. The **P** command alternatively turns this mode on and off; it is initially on if the **–p** option is specified, otherwise off. The current line number is unchanged.

Quit Command

Synopsis: `q`

The **q** command causes *ed* to exit. If the buffer has changed since the last time the entire buffer was written, the user will be warned, as described previously.

Quit Without Checking Command

Synopsis: `Q`

The **Q** command causes *ed* to exit without checking if changes have been made in the buffer since the last **w** command.

Read Command

Synopsis: `($)r[`*`file`*`]`

The **r** command reads in the file named by the pathname *file* and appends it after the addressed line. If no *file* argument is given, the currently remembered pathname, if any, will be used (see **e** and **f** commands). The currently remembered pathname will not be changed unless there is no remembered pathname. Address 0 is valid for **r** and causes the file to be read at the beginning of the buffer. If the read is successful, and **–s** was not specified, the number of bytes read will be written to standard output in the following format:

`"%d\n", <`*`number of bytes read`*`>`

The current line number will be set to the address of the last line read in. If *file* is replaced by **!**, the rest of the line will be taken to be a shell command line whose output is to be read. Such a shell command line will not be remembered as the current pathname.

Substitute Command

Synopsis: `(.,.)s/`*`RE`*`/`*`replacement`*`/`*`flags`*

The **s** command searches each addressed line for an occurrence of the specified RE and replace either the first or all (non-overlapped) matched strings with the *replacement*; see the following description of the **g** suffix. It is an error if the substitution fails on every addressed line. Any character other than space or newline can be used instead of a slash to delimit the *RE* and the replacement. Within the *RE*, the *RE* delimiter itself can be used as a literal character if it is preceded by a backslash. The current line will be set to the address of the last line on which a substitution occurred.

An ampersand (&) appearing in the *replacement* will be replaced by the string matching the RE on the current line. The special meaning of & in this context can be suppressed by preceding it by backslash. As a more general feature, the characters `\`*n*, where *n* is a digit, will be replaced by the text matched by the corresponding back-reference expression. When the character % is the only character in the *replacement*, the *replacement* used in the most recent substitute command will be used as the *replacement* in the current substitute command; if there was no previous substitute command, the use of % in this manner is an error. The % loses its special meaning when it is in a replacement string of more than one character or is preceded by a backslash. For each backslash (`\`) encountered in scanning *replacement* from beginning to end, the following character loses its special meaning (if any). It is unspecified what special meaning is given to any character other than &, `\`, % or digits.

A line can be split by substituting a newline character into it. The application must escape the newline character in the *replacement* by preceding it by backslash. Such substitution cannot be done as part of a **g** or **v** command list. The current line number will be set to the address of the last line on which a substitution is performed. If no substitution is performed, the current line number is unchanged. If a line is split, a substitution is considered to have been performed on each of the new lines for the purpose of determining the new current line number. A substitution is considered to have been performed even if the replacement string is identical to the string that it replaces.

The value of *flags* must be zero or more of:

count Substitute for the *count*th occurrence only of the *RE* found on each addressed line.

g Globally substitute for all non-overlapping instances of the *RE* rather than just the first one. If both **g** and *count* are specified, the results are unspecified.

l Write to standard output the final line in which a substitution was made. The line will be written in the format specified for the **l** command.

n Write to standard output the final line in which a substitution was made. The line will be written in the format specified for the **n** command.

p Write to standard output the final line in which a substitution was made. The line will be written in the format specified for the **p** command.

Copy Command

Synopsis: `(.,.)t`*address*

The **t** command is equivalent to the **m** command, except that a copy of the addressed lines will be placed after address *address* (which can be 0); the current line number will be set to the address of the last line added.

Undo Command

Synopsis: `u`

The **u** 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**, **u**, **v**, **G** or **V** command. All changes made to the buffer by a **g**, **G**, **v** or **V** global command will be undone as a single change; if no changes were made by the global command (such as with **g/***RE***/p**), the **u** command will have no effect. The current line number will be set to the value it had immediately before the command being undone started.

Global Non-matched Command

Synopsis: `(1,$)v/`*RE*`/`*command list*

This command is equivalent to the global command **g** except that the lines that are marked during the first step will be those that do not match the *RE*.

Interactive Global Not-matched Command

Synopsis: `(1,$)V/`*RE*`/`

This command is equivalent to the interactive global command **G** except that the lines that are marked during the first step will be those that do not match the *RE*.

Write Command

Synopsis: `(1,$)w[`*file*`]`

The **w** command writes the addressed lines into the file named by the pathname *file*. The command will create the file, if it does not exist, or will replace the contents of the existing file. The currently remembered pathname will not be changed unless there is no remembered pathname. If no pathname is given, the currently remembered pathname, if any, will be used (see the **e** and **f** commands); the current line number is unchanged. If the command is successful, the number of bytes written will be written to standard output, unless the **–s** option was specified, in the following format:

`"%d\n", <`*number of bytes written*`>`

If *file* begins with !, the rest of the line will be taken to be a shell command line whose standard input will be the addressed lines. Such a shell command line will not be remembered as the current pathname. This usage of the write command with ! is be considered as a ‘‘last **w**

command that wrote the entire buffer'', as described previously; thus, this alone will not prevent the warning to the user if an attempt is made to destroy the editor buffer via the **e** or **q** commands.

Line Number Command

Synopsis: `($)=`

The line number of the addressed line will be written to standard output in the following format:

`"%d\n", <`*`line number`*`>`

The current line number is unchanged by this command.

Shell Escape Command

Synopsis: `!`*`command`*

The remainder of the line after the **!** will be sent to the command interpreter to be interpreted as a shell command line. Within the text of that shell command line, the unescaped character **%** will be replaced with the remembered pathname; if a **!** appears as the first character of the command, it will be replaced with the text of the previous shell command executed via **!**. Thus, **!!** will repeat the previous **!***command*. If any replacements of **%** or **!** are performed, the modified line will be written to the standard output before *command* is executed. The **!** command will write:

`"!\n"`

to standard output upon completion, unless the **–s** option is specified. The current line number is unchanged.

Null Command

Synopsis: `(.+1)`

An address alone on a line causes the addressed line to be written. A newline character alone is equivalent to .+1p. The current line number will be set to the address of the written line.

EXIT STATUS

The following exit values are returned:

0 Successful completion without any file or command errors.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When an error in the input script is encountered, or when an error is detected that is a consequence of the data (not) present in the file or due to an external condition such as a read or write error:

- If the standard input is a terminal device file, all input will be flushed, and a new command read.
- If the standard input is a regular file, *ed* will terminate with a non-zero exit status.

APPLICATION USAGE

Because of the extremely terse nature of the default error messages, the prudent script writer will begin the *ed* input commands with an **H** command, so that if any errors do occur at least some clue as to the cause will be made available.

EXAMPLES

None.

FUTURE DIRECTIONS

The obsolescent single-minus form may be withdrawn from a future issue. Applications should use the **–s** option.

SEE ALSO

ex, *sed*, *sh*, *vi*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

egrep – search a file with an ERE pattern

SYNOPSIS

OB `egrep [ -c| -l][-inv] -e pattern_list [file...]`

OB `egrep [ -c| -l][-inv] -f pattern_list [file...]`

OB `egrep [ -c| -l][-inv] pattern_list [file...]`

DESCRIPTION

The name *egrep* is an obsolescent version equivalent to *grep* –**E**.

A command invoking the *egrep* utility with the –**e** option specified is equivalent to the command:

```
grep -E [ -c| -l][-inv] -e pattern_list [file...]
```

A command invoking the *egrep* utility with the –**f** option specified is equivalent to the command:

```
grep -E [ -c| -l][-inv] -f pattern_list [file...]
```

A command invoking the *egrep* utility with neither the –**e** nor the –**f** option specified is equivalent to the command:

```
grep -E [ -c| -l][-inv] pattern_list [file...]
```

APPLICATION USAGE

Unlike *grep* –**E**, multiple –**e** or –**f** options produce undefined results. Adjacent newline characters in the *pattern* operand or –**e** *pattern_list* option-argument also produce undefined results.

FUTURE DIRECTIONS

The *egrep* utility may be withdrawn from a future issue; applications should migrate to *grep* –**E**.

SEE ALSO

grep.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions used by *egrep* have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Separated from the *fgrep* description.

NAME

env – set environment for command invocation

SYNOPSIS

```
env [-i][name=value]... [utility [argument..]]
```

OB

```
env [-][name=value]... [utility [argument..]]
```

DESCRIPTION

The *env* utility will obtain the current environment, modify it according to its arguments, then invoke the utility named by the *utility* operand with the modified environment.

Optional arguments will be passed to *utility*.

If no *utility* operand is specified, the resulting environment will be written to the standard output, with one *name=value* pair per line.

OPTIONS

OB The *env* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except for its non-standard usage of –.

The following options are supported:

–i

OB **–**

Invoke *utility* with exactly the environment specified by the arguments; the inherited environment will be ignored completely.

OPERANDS

The following operands are supported:

name=value

Arguments of the form *name=value* modify the execution environment, and are placed into the inherited environment before the *utility* is invoked.

utility The name of the utility to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

argument

A string to pass as an argument for the invoked utility.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *env*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the location of the *utility*, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**. If *PATH* is specified as a *name=value* operand to *env*, the *value* given will be used in the search for *utility*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If no *utility* operand is specified, each *name=value* pair in the resulting environment will be written in the form:

```
"%s=%s\n", <name>, <value>
```

If the *utility* operand is specified, the *env* utility will not write to standard output.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
If the *utility* utility is invoked, the exit status of *env* will be the exit status of *utility;* otherwise, the *env* utility will exit with one of the following values:

0 The *env* utility completed successfully.

1-125 An error occurred in the *env* utility.

126 The utility specified by *utility* was found but could not be invoked.

127 The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The *command*, *env*, *nice*, *nohup*, *time* and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

Historical implementations of the *env* utility use the **XSH** specification *execvp*( ) or *execlp*( ) functions to invoke the specified utility; this provides better performance and keeps users from having to escape characters with special meaning to the shell. Therefore, shell functions, special built-ins and built-ins that are only provided by the shell are not found.

EXAMPLES

The following command:

```
env -i PATH=/mybin mygrep xyz myfile
```

invokes the command *mygrep* with a new *PATH* value as the only entry in its environment. In this case, *PATH* is used to locate *mygrep*, which then must reside in **/mybin**.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.5 on page 27.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

ex – text editor

SYNOPSIS

EX `ex [-rR][-l][-s | -v ][-c command]-t tagstring][-w size][file...]`

OB `ex [-rR][-l][-s | -v ][+command]-t tagstring][-w size][file...]`

DESCRIPTION

The *ex* utility is a line-oriented text editor that supports both line and full-screen editing (see *vi*).

Certain block-mode terminals do not have all the capabilities necessary to support the complete *ex* definition, such as the full-screen editing commands (*visual* mode) or *open mode*. When these commands cannot be supported on such terminals, this condition will not produce an error message such as ‘‘not an editor command’’ nor report a syntax error. The implementation may either accept the commands and produce results on the screen that are the result of an unsuccessful attempt to meet the requirements of this document or report an error describing the terminal-related deficiency. The affected commands are noted as they occur later in this section.

OPTIONS

The *ex* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c *command*

OB *+command*

Begin editing by executing the specified *ex* command-mode commands. As with normal editing command-line entries, the *command* option-argument can consist of multiple *ex* commands separated by vertical-line characters (|). The use of commands that enter input or visual modes in this manner produces undefined results.

EX **–l** (The letter ell.) Set lisp mode; indents appropriately for Lisp code; the **()**, **{}**, **[[** and **]]** commands in visual mode are modified to have meaning for Lisp.

–r Recover the named files after an editor or system crash, after the editor has been terminated by a signal, or after the use of a **preserve** editor command. A *crash* in this context is an unexpected failure of the system or utility that requires restarting the failed system or utility. A system crash implies that any utilities running at the time also crash. In the case of an editor or system crash, the degree of recovery (the number of changes to the buffer since the most recent **preserve** command) available is unspecified.

If no *file* operands are given, all other options, the *EXINIT* variable and any **.exrc** files will be ignored; a list of all recoverable files available to the invoking user will be written; and *ex* will exit without reading files or processing user commands.

–R Set read-only mode, preventing accidental overwriting of the files. Any command that would write to a file will require the ! suffix (see, for example, the **write** command) to be effective in this mode.

–s

OB – Prepare *ex* for batch use by taking the following actions:

- Suppress writing prompts and informational (but not diagnostic) messages.

- Ignore the value of *TERM* and any implementation default terminal type and assume the terminal is a type incapable of supporting visual mode; see the **visual** command and the description of *vi*.
- Suppress the use of the *EXINIT* environment variable and the reading of any **.exrc** file (see **EXTENDED DESCRIPTION**).

–t *tagstring*
Edit the file containing the specified *tagstring* and proceed as if the first command were **:tag** *tagstring*. (See *ctags*.) The tags feature represented by **–t** *tagstring* and the **ta** command are optional. It is provided on any system that also provides a conforming implementation of *ctags*; otherwise, the use of **–t** produces undefined results.

–v Invoke *vi*.

–w *size*
Set the value of the *window* editor option to *size*.

OB If both the **–t** *tagstring* and **–c** *command* (or the obsolescent *+command*) options are given, the **–t** *tagstring* will be processed first; that is, the file containing the tag is selected by **–t** and then the command is executed.

OPERANDS

The following operand is supported:

file A pathname of a file to be edited.

STDIN

The standard input must be a text file consisting of commands, as described in **EXTENDED DESCRIPTION**.

INPUT FILES

Input files must be text files or files that would be text files except for an incomplete last line that is not longer than {LINE_MAX}–1 bytes in length and contains no NUL characters. The editing of other forms of files may optionally be allowed by *ex* implementations.

The **.exrc** files (see **EXTENDED DESCRIPTION**) must be text files consisting of commands.

By default, *ex* reads lines from the files to be edited without interpreting any of those lines as any form of editor command.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ex*:

COLUMNS
Override the system-selected horizontal screen size. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

EXINIT Determine a list of *ex* commands that are executed on editor start-up, before reading the first file. The list can contain multiple commands by separating them using a vertical-line (|) character. See **EXTENDED DESCRIPTION** for more details of the initialisation phase.

HOME Determine a pathname of a directory that will be searched for an editor start-up file named **.exrc**; see **EXTENDED DESCRIPTION** for details.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the behaviour of character classes within regular expressions, the classification of characters as upper- or lower-case letters, the case conversion of letters, and the detection of word boundaries.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LINES Override the system-selected vertical screen size, used as the number of lines in a screenful and the vertical screen size in visual mode. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path for the shell command specified in the editor commands **shell**, **read** and **write**; see the description of command search and execution in **Command Search and Execution** on page 47.

SHELL Determine the preferred command-line interpreter for use in !, **shell**, **read** and other commands with an operand of the form !*string*. For the **shell** command, the program will be invoked with the single argument –**i**, for all others it will be invoked with the two arguments –**c** and *string*. If no *SHELL* environment variable is set, or it is set to a null string, the *sh* utility will be used.

TERM Determine the name of the terminal type. If this variable is unset or null, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS

The following actions will be taken upon receipt of signals:

SIGINT When an interrupt occurs, *ex* will alert the terminal and write a message. The current editor command will be aborted and *ex* will return to the command level and prompt for another command. If the standard input is not a terminal device, *ex* will exit at the interrupt and return a non-zero exit status. (The alerting action can be modified by the use of the **errorbells** editor option; see **Edit Options in ex** on page 317.)

SIGCONT
The screen will be refreshed (if in visual mode).

SIGHUP
If the current buffer has changed since the last **e** or **w** command, *ex* will attempt to save the current file in a state such that it can be recovered later by an *ex* –**r** command.

The action taken for all other signals is unspecified.

STDOUT

The standard output is used only for writing prompts to the user, for informational messages and for writing lines from the file.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output from *ex* must be text files that are identical to the input files if no changes have been made to the files by commands, with the exception that in all cases where a forced session termination (the *ex* command q!) has not been issued prior to any file write, a trailing newline character will be added to the last line of the file if one was not present in the input.

EXTENDED DESCRIPTION

EX

The pathname of the file being edited by *ex* is referred to as the *current* file. The text of the file is read into a working version of the file (called *buffer* in this section; intermediate versions of the buffer may be kept in a file that is created in the directory indicated by the **directory** editor option) and all editing changes are performed on that version; the changes have no effect on the original file until an *ex* command causes the file to be written out. Lines in the buffer may each be limited to {LINE_MAX} bytes and an error message may be written if the limit is exceeded during editing.

The *alternative* pathname is the name of the last file mentioned in an editor command, or the previous current pathname if the last file mentioned became the current file. When the character % appears in a pathname entered as part of a command argument, it is replaced by the current pathname; the character # is replaced by the alternative pathname. Any character, including % and #, retains its literal value (is escaped) when preceded by a backslash.

When an error occurs, *ex* will alert the terminal and write a message. (The alerting action can be modified by the use of the **errorbells** editor option.)

If the system crashes, *ex* will attempt to preserve the buffer if any unwritten changes were made. The command-line option **–r** can be used to retrieve the saved changes.

During initialisation, before the first file is read or any user commands from the terminal are processed, if the environment variable *EXINIT* is set, the editor will execute the *ex* commands contained in that variable. If the variable is not set, *ex* will attempt to read commands from the file **$HOME/.exrc** (the file **.exrc** in the directory referred to by the *HOME* environment variable). If and only if *EXINIT* or **$HOME/.exrc** sets the editor option **exrc**, *ex* finally will attempt to read commands from a file **.exrc** in the current directory. In the event that *EXINIT* is not set and the current directory is the user's home directory, any **.exrc** file will only be processed once. No **.exrc** file will be read unless it is owned by the same user ID as the effective user ID of the process. After any **.exrc** files are processed, any commands specified by the **–c** option will be processed.

By default, *ex* starts in the command mode, which is indicated by the **:** prompt. The input mode can be entered by **append**, **insert** or **change** commands; it can be exited (and command mode reentered) by typing a period (.) alone at the beginning of a line. There is one other mode, visual mode, in which full-screen editing is available. This is described more fully under the **visual** command and in the *vi* utility description. The command line can consist of multiple *ex* commands separated by vertical-line characters (|). The use of commands that enter input or visual modes in this manner, unless they are the final command on the line, produces undefined results.

Command lines beginning with the double-quote character (") are ignored. This can be used for comments in an editor script.

Addressing in ex

Addressing in *ex* relates to the *current line*. In general, 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. When the buffer contains no lines, the current line is set to zero.

Addresses are constructed by one of the following methods:

1. The address . (period) refers to the current line.
2. The address $ refers to the last line of the buffer.
3. The address *n*, where *n* is a decimal number, refers to the *n*th line of the buffer.
4. The address *'x* refers to the line marked with the mark-name character *x*, which must be a lower-case letter of the POSIX locale. Lines can be marked with the **ma** or k commands described below.
5. A regular expression (RE) enclosed by slashes (/) is an address, and refers to 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. The second slash can be omitted at the end of a command line. If the **wrapscan** option is set, the search will wrap around to the beginning of the buffer and continue up to and including the current line, so that the entire buffer is searched.
6. An RE enclosed in question marks (?) addresses the first line found by searching backward from the line preceding the current line toward the beginning of the buffer and stopping at the first line containing a string matching the RE. The second question mark can be omitted at the end of a command line. If the **wrapscan** option is set, the search will wrap around from the beginning of the buffer to the end of the buffer and continue up to and including the current line, so that the entire buffer is searched.
7. An address followed by a plus sign (+) or a minus sign (–) followed by a decimal number is an offset address, and refers to the first address plus (respectively minus) the indicated number of lines. If the address is omitted, the addition or subtraction is taken with respect to the current line.
8. An address of + or – followed by a number is taken with respect to the current line number; for example, –5 is understood to mean .–5.
9. An address ending with + or – has 1 added to or subtracted from the address, respectively. As a consequence of this rule and of rule 8 above, the address – refers to the line preceding the current line. Moreover, trailing + and – characters have a cumulative effect; for example, – – refers to the current line less 2.
10. A percent sign (%) stands for the address pair 1,$.

Commands require zero, one or two addresses. See the descriptions of *line* and *range* in **Command Descriptions in ex** on page 304. Commands that require zero addresses regard the presence of an address as an error.

Adjacent addresses in a *range* must be separated from each other by a comma (,) or 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 corresponds to a line that follows, in the buffer, the line corresponding to the first address. The first address must be less than or equal to the second address. The first address must be greater than or equal to the first line of the editing buffer and the last address must be less than or equal to the last line of the editing buffer. Any other case is an error.

All of the following examples are valid *addresses*:

+++ three lines after the current line

/*re*/– one line before the next occurrence of *re*

–2 two lines before the current line.

Command Descriptions in ex

The following symbols are used in this section to represent optional modifiers. Any or all can be omitted; the defaults are shown.

line A single line address, given in any of the forms described in **Addressing in ex** on page 303; the default for *line* is the current line.

range A *line*, or a pair of line addresses, separated by a comma or semicolon (see **Addressing in ex** on page 303 for the difference between the two); the default for *range* is the current line only (.,.). A percent sign (%) stands for the range (1,$). If the range specified is such that the starting address exceeds the ending address, the range is invalid and the command will not be performed. If more than the expected number of addresses are given in a range, the greatest valid number of the last ones given will be used. For example, 1,3,5p prints lines 3 to 5, inclusive (because two is the greatest valid number in the range accepted by **print**).

count A positive integer, specifying the number of lines to be affected by the command; the default for *count* is 1.

flags One of the characters #, p or l (ell), or both # and l to add numbers to list-format output. When a command with such a flag completes, the addressed lines will be written out as if by the corresponding **#**, **p** or **l** command. The use of *flags* applies to all lines written by the **list**, **number**, **open**, **print**, **substitute**, **visual**, **&** and z commands; for other commands, it applies to the current line at the completion of the command. In addition, any number of + or – characters can also be given after the flags, in which case the line written is not the one affected by the command, but rather the line addressed by the offset address as described above. The default for *flags* is null.

buffer One of a number of named areas for saving text. The named buffers are specified by the lower-case letters of the POSIX locale. Specifying *buffer* causes the area of text affected by the command to be stored into the buffer as it was before the command took effect. This argument is also used on the **put** command and the visual mode put commands (**p** and **P**) to specify the buffer that will provide the text to insert.

If the buffer name is specified in upper-case, and the buffer is to be modified (as with a deletion or yanking command), the buffer will be appended to rather than being overwritten. If the buffer is not to be modified (as in a visual mode **put** command) the buffer name can be specified in lower-case or upper-case with the same results. There is also one unnamed buffer, which is the repository for all text deleted (with the **delete** or visual mode **d** command) or yanked (with the **yank** or visual mode **y** command) when no buffer is specified.

There are also numbered buffers, 1 to 9, inclusive, which are accessible only from visual mode. These buffers are special in that, in visual mode, when deleted text is placed in the unnamed buffer, it also is placed in buffer 1, the previous contents of buffer 1 are placed in buffer 2, and so on. Any text in buffer 9 will be lost. Text that is yanked (or otherwise copied) into the unnamed buffer does not modify the numbered buffers. Text cannot be placed directly into the numbered buffers although it can be retrieved from them by using a visual mode **put** command with the buffer name given as a

number. When the *buffer* modifier is not used in the commands below, the unnamed buffer is the default.

file A pattern used to derive a pathname; the default is the current file, as defined above. If no current file has yet been established, a warning will be written and the command will be aborted, except where specifically noted in the individual command descriptions that follow. The pattern will be subjected to the process of shell word expansions (see Section 2.6 on page 31); if more than a single pathname results and the command is expecting one file, the effects are unspecified.

word In the POSIX locale, a *word* consists of a maximal sequence of letters, digits and underscores, delimited at both ends by characters other than letters, digits or underscores, or by the beginning or end of a line or the file.

! A character that can be appended to the command to modify its operation, as detailed in the individual command descriptions.

If both a *count* and a *range* are specified for a command that uses them, the number of lines affected will be taken from the *count* value rather than the *range*. The starting line for the command is taken to be the first line addressed by the *range*.

When only a *line* or *range* is specified with no command, the implied command is either a **print**, **list** or **number** (**p**, **l** or **#**). The command selected will be the last of these three commands to be used, including use as a *flag*. When no range or count is specified and the command line is a blank line, the current line will be written, and the current line will be set to .+1.

Zero or more blank characters can precede or follow the addresses, *count*, *flags* or command name. Any object following a command name (such as *buffer*, *file* and so forth) that begins with an alphabetic character will be separated from the command name with at least one blank character.

For each of the commands listed below, the command can be entered as the abbreviation (those characters in the Synopsis command word preceding the **(),** the full command (all characters shown for the command word, omitting the **[** and **]),** or any subset of the characters of the full command down to the abbreviation. For example, the **args** command (shown as **ar[gs]** in the Synopsis) can be entered as **ar**, **arg** or **args**.

Abbreviate

Synopsis: `ab[brev] word rhs`

Add the named abbreviation to the current abbreviation list. In visual mode, if *word* is typed so that it is preceded and followed by characters that cannot be part of a *word* (as defined previously), it will be replaced by the string *rhs*.

Append

Synopsis: `[line] a[ppend][!]`

Enter input mode; the input text will be placed after the specified line. If line 0 is specified, the text will be placed at the beginning of the buffer. The current line indicator will be set to the last input line; if no lines are input, it will be set to the target line, or to the first line of the file if a target of 0 was specified. Following the command name with **!** causes the **autoindent** editor option setting to be toggled for the duration of this command only.

Arguments

Synopsis: `ar[gs]`

Write the argument list with the current argument inside [and]. The argument list is the list of operands on start-up, which can subsequently be replaced by the operands of the **next** command.

Change

Synopsis: `[`*range*`] c[hange][!] [`*count*`]`

Enter input mode; the input text will replace the specified lines (*range*). The current line indicator will be set to the last line input; if no lines were input, it will be set to the line before the target line, or to the first line of the file if there are no lines preceding the target. Following the command name with ! causes the **autoindent** editor option setting to be toggled for the duration of this command only.

Change Directory

Synopsis: `chd[ir][!] [`*directory*`]`
Synopsis: `cd[!] [`*directory*`]`

Change the current working directory to *directory*. The argument will be subjected to the process of shell word expansions (see Section 2.6 on page 31). When invoked with no *directory* argument, and the *HOME* environment variable is set to a non-empty value, the directory name in the *HOME* environment variable will become the new working directory. If *HOME* is empty or undefined, the default behaviour is implementation-dependent. If the current buffer has been modified since the last write, a warning will be written and the command will fail. This warning can be overridden by appending ! to the command name, which will allow the command to complete.

Copy

Synopsis: `[`*range*`] co[py] `*line*` [`*flags*`]`
Synopsis: `[`*range*`] t `*line*` [`*flags*`]`

Place a copy of the specified lines (*range*) after the specified destination *line*; line 0 specifies that the lines will be placed at the beginning of the buffer.

Delete

Synopsis: `[`*range*`] d[elete] [`*buffer*`] [`*count*`] [`*flags*`]`

Delete the specified lines from the buffer. If a named *buffer* is specified, the deleted text will be saved in it; otherwise, the deleted text will be saved in the unnamed buffer. If the command name is followed by a letter that could be interpreted as either a buffer name or a *flag* value (because neither a *count* nor an additional *flags* value was given), *ex* will consider the letter to be a *flags* value if the letter directly follows the command name, without any blank character separation; if the letter is preceded by one of more blank characters, it is considered a buffer name.

For example:

1dp or **1deletep** Deletes the first line and prints the line that was second.

1d p Deletes the first line, saving it in buffer p.

1d p1l (Pee-one-ell.) Deletes the first line, saving it in buffer p, and listing the line that was second.

The current line indicator will be set to the line following the deleted lines, or to the last line if the deleted lines were at the end.

Edit

Synopsis: `e[dit][!] [+`*`line`*`] [`*`file`*`]`
Synopsis: `ex[!] [+`*`line`*`] [`*`file`*`]`

Begin editing *file*. If the current buffer has been modified since the last write, a warning will be written and the command will be aborted. This action can be overridden by appending the character ! to the command name (e! *file*). The current line indicator will be affected as follows:

- If *file* is omitted or results in the current file, the current line indicator will not be changed.
- Otherwise, the current line indicator will be the last line of the buffer; however, if this command is executed from within visual mode, the current line indicator will be the first line of the buffer.

If the *+line* option is specified, the current line indicator will be set to the specified position, where *line* can be a number (or $) or specified as ''*/pattern*'' or ''**?***pattern*''. Preceding the pattern with a / will start a search from the beginning of the file. Preceding the pattern with a ? will start a search from the end of the file. This command is affected by the editor options **autowrite** and **writeany**.

File

Synopsis: `f[ile] [`*`file`*`]`

Write the current pathname, the number of lines, and the current position when no *file* argument has been specified; *ex* may write other unspecified information. If no current file has yet been established, an unspecified message will be written to indicate that no file is being edited. With *file*, *ex* will change the current filename to *file* without changing the contents of the buffer or the previous current file.

Global

Synopsis: `[`*`range`*`] g[lobal] /`*`pattern`*`/ [`*`commands`*`]`
Synopsis: `[`*`range`*`] v /`*`pattern`*`/ [`*`commands`*`]`

Mark the lines within the given range that match (g) or do not match (v) the given pattern. Then execute the *ex* commands given by *commands* with the current line (.) set to each marked line. The *range* defaults to the entire file.

Multiple *commands* can be specified, one per line, by escaping each newline character with a backslash. If *commands* are omitted, each line will be written. For the **append**, **change** and **insert** commands, the input text will be included as part of the **global** command line; in this case the terminating period can be omitted if it ends *commands*. The **visual** command can be specified as one of the *commands*. In this mode, input will be taken from the terminal. Entering a Q from visual mode causes the next line matching the pattern to be selected and visual mode to be reentered, until the list is exhausted.

The **global** command itself and the **undo** command cannot be used in *commands*. The editor options **autoprint**, **autoindent** and **report** will be inhibited for the duration of the **g** or **v** command.

Insert

Synopsis: **[***line***]** i**[**nsert**][**!**]**

Enter input mode; the input text will be placed before the specified line. The current line indicator will be set to the last line input; if no lines were input, it will be set to the line before the target line, or to the first line of the file if there are no lines preceding the target. Following the command name with the character ! causes the **autoindent** editor option setting to be toggled for the duration of this command only.

Join

Synopsis: **[***range***]** j**[**oin**][**!**] [***count***] [***flags***]**

Join the text from the specified lines together into one line. In the POSIX locale, when the last character on the first line of a pair of lines to be joined is a period, two space characters will be added following the period; when the last character of the first line is a blank character or when the first character on the second line of the pair is a), no space characters will be added; otherwise, one space character will be added following the last character of the first line. Extra blank characters at the start of a line will be discarded.

Appending a character ! to the **join** command name causes a simpler join with no white-space processing, independent of the current locale.

List

Synopsis: **[***range***]** l**[**ist**] [***count***] [***flags***]**

Write the addressed lines in a way that should be unambiguous: non-printable characters will be written as implementation-dependent multi-character sequences; the end of the line will be marked with a $.

Long lines will be folded; the length at which folding occurs is unspecified, but should be appropriate for the output device. The only useful flag is #, for line numbers. The current line indicator will be set to the last line written.

Map

Synopsis: map**[**!**] [***x rhs***]**

EX Define macros for use in visual mode. The first argument must be a single character or the sequence *#digit* (the latter meaning one of the terminal's numbered function keys). When this character or function key is typed in visual mode, the action will be as if the corresponding *rhs* had been typed. If the character ! is appended to the command name **map**, the mapping is effective during input mode rather than command mode. This allows *x* to have two different macro definitions at the same time: one for command mode and one for input mode. Non-printable characters, except for a tab character, require escaping with a <control>-V (or <control>-Q) to be entered in the arguments. On certain block-mode terminals, the mapping need not occur immediately (for example, it may occur after the terminal transmits a group of characters to the system), but it will achieve the same results of modifying the file as if it occurred immediately. Implementations may restrict the set of commands accepted within *rhs*; the list of restrictions is implementation-dependent.

The **map** command with no arguments will write all of the macros currently defined. If ! is appended to the command, only the macros effective during input mode will be written; otherwise, only the macros effective during command mode will be written.

Mark

Synopsis: **[***line***]** ma**[**rk**]** *x*
Synopsis: **[***line***]** k *x*

Give the specified line the specified mark *x*, which must be a single lower-case letter of the POSIX locale. The current line position will not be affected. The expression *'x* can then be used as an address in any command requiring one. For example ''**.,'***x***d**'' deletes all the lines from the current one to the marked line. Also see the *vi* '' and '' commands for uses of the mark in visual mode. If the *'x* command is used in non-visual mode, the character marked will be the first non-blank character of the current line. Otherwise, the character marked will be the character at the current column of the current line.

Move

Synopsis: **[***range***]** m**[**ove**]** *line*

Move the specified lines (*range*) to be after the target line (*line*). The current line indicator will be set to the first of the moved lines.

Next

Synopsis: n**[**ext**][**!**] [***file ...***]**

Edit the next file from the argument list. If the current buffer has been modified since the last write, a warning will be written and the command will be aborted. This action can be overridden by appending the character ! to the command name (n!). The argument list can be replaced by specifying a new one as operands to this command. Editing then starts with the first file on this new list. The current line indicator will be reset as described for the **edit** command. This command is affected by the editor options **autowrite** and **writeany**; see **Edit Options in ex** on page 317 for details.

Number

Synopsis: **[***range***]** nu**[**mber**] [***count***] [***flags***]**
Synopsis: **[***range***]** # **[***count***] [***flags***]**

Write the selected lines, each preceded with its line number in decimal. Non-printable characters, except for a tab character, will be expanded as specified by the **print** command. The format is as follows:

```
"%+6d\t%s", <line number>, <line text>
```

The only meaningful flag is l, which causes the additional expanded writing of tab characters and end-of-lines done by the **list** command to be performed. The current line indicator will be set to the last line written.

Open

Synopsis: **[***line***]** o**[**pen**]** */pattern/* **[***flags***]**

Enter open mode, which is equivalent to visual mode with a one-line window. All the visual mode commands are available. If a match is found for the optional regular expression in *line*, the cursor will be placed at the start of the matching pattern. The visual mode command **Q** (see *vi*) will exit open mode. This command need not be supported on block-mode terminals.

Preserve

Synopsis: `pre[serve]`

Save the current buffer in a form that can later be recovered by using *ex* –**r** or by using the **recover** command. After the file has been preserved, a mail message will be sent to the user. This message can be read by invoking the *mailx* utility. The message will contain the name of the file, the time of preservation, and an *ex* command that could be used to recover the file. Additional information may be included in the mail message.

Print

Synopsis: `[`*range*`] p[rint] [`*count*`] [`*flags*`]`

Write the addressed lines. Non-printable characters, except for the tab character, will be written as implementation-dependent multi-character sequences.

Long lines will be folded; the length at which folding occurs is unspecified, but should be appropriate for the output device. The only meaningful *flags* are # and l. The current line indicator will be set to the last line written.

Put

Synopsis: `[`*line*`] pu[t] [`*buffer*`]`

Put back deleted or yanked lines after line *line*. A buffer can be specified; otherwise, the text in the unnamed buffer (where deleted or yanked text is placed by default) will be restored. The current line indicator will be set to the first line put back.

Quit

Synopsis: `q[uit][!]`

Terminate the editing session. If the current buffer has been modified since the last write, a warning will be written and the command will fail. This warning can be overridden and an exit forced, discarding changes, by appending the character ! to the command name.

Read

Synopsis: `[`*line*`] r[ead][!] [`*file*`]`

Place a copy of the specified file in the current buffer after the target line (which can 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* will become the current file. If there is no current file nor *file* operand, the command will fail.

The current line indicator will be set to the last line read. In visual mode, the current line indicator will be set to the first line read. If *file* is preceded by !, *file* is taken to be an operating system command and passed to the program named in the *SHELL* environment variable; the resultant output will be read in to the buffer. The special meaning of ! can be overridden by escaping it with a backslash character.

Recover

Synopsis: `rec[over] `*file*

Attempt to recover *file* if it was saved as the result of a **preserve** command, the receipt of a signal (see **ASYNCHRONOUS EVENTS**), or a system or editor crash. The current line indicator will be reset as described for the **read** editor command.

Rewind

Synopsis: `rew[ind][!]`

Rewind the argument list; that is, the current file will be set to the first file in the argument list. This is equivalent to a **next** command with the current argument list as its operands. If the current buffer has been modified since the last write, a warning will be written and the command will be aborted. The action can be overridden by appending the character ! to the command name (rew!). The current line indicator will be reset as described for the **read** editor command. This command is affected by the editor options **autowrite** and **writeany**; see **Edit Options in ex** on page 317 for details.

Set

Synopsis: `se[t] [`*`option`*`[=[`*`value`*`]]`...`] [no`*`option`*`...] [`*`option`*`?...] [all]`

When no arguments are specified, write those options whose values have been changed from the default settings; when the argument **all** is specified, write all of the option values.

Giving an option name followed by the character ? causes the current value of that option to be written. The ? can be separated from the option name by zero or more blank characters. The ? is necessary only for Boolean valued options. Boolean options can be given values by the form **se** *option* to turn them on or se no*option* to turn them off; string and numeric options can be assigned by the form se *option=value*. Blank characters in strings can be included as is by preceding each such character with a backslash. More than one option can be set or listed by a single **set** command by specifying multiple arguments, each separated from the next by one or more blank characters.

See **Edit Options in ex** on page 317 for further details about options.

Shell

Synopsis: `sh[ell]`

Invoke the program named in the *SHELL* environment variable with the argument **–i** (interactive mode). Editing will be resumed when the program exits.

Source

Synopsis: `so[urce] `*`file`*

Read and execute commands from the file specified by the mandatory argument *file*. Such **so** commands can be nested. The maximum supported nesting depth is implementation-dependent, but will be at least one.

Substitute

Synopsis: `[`*`range`*`] s[ubstitute] [/`*`pattern`*`/`*`repl`*`/[`*`options`*`] [`*`count`*`] [`*`flags`*`]]`

Replace the first instance of the pattern *pattern* by the string *repl* on each specified line. (See **Regular Expressions in ex** on page 316 and **Replacement Strings in ex** on page 316.) If the */pattern/repl/* argument is not present, the */pattern/repl/* from the previous **substitute** command will be used. If *options* includes the letter **g** (global), all non-overlapping instances of the pattern in the line will be substituted. If the option letter **c** (confirm) is included, then before each substitution the line will be written with ^ characters written on the following line, adjacent to and identifying the pattern to be replaced; an affirmative response causes the substitution to be done, while any other input aborts it. An affirmative response consists of a line with the affirmative response (as defined by the current locale) at the beginning of the line. Such a line

will be subject to editing in the same way as the command line (the / or : line at the bottom of the screen). The current line indicator will be set to the last line substituted. When the **c** option is used, typing the interrupt character or receiving the SIGINT signal will stop the substitute operation and *ex* will return to command mode. All substitutions completed before the interrupt occurred will be retained, and none will be made after that point. The current line indicator will be set to the last line substituted. This command is affected by the *LC_MESSAGES* environment variable and the **wrapscan** option.

Suspend

Synopsis: `su[spend][!]`
Synopsis: `st[op][!]`

Allow control to return to the invoking process; *ex* will suspend itself as if it had received the
JC SIGTSTP signal. The suspension will occur only if job control is enabled in the invoking shell (see the description of *set* –**m**).

Following either **suspend** or **stop** with the character ! will affect the operation of the **autowrite** editor option for this command only.

The current **susp** character (see *stty*) will also cause the suspension.

Tag

Synopsis: `ta[g][!]` *tagstring*

Search for the *tagstring*, which can be in a different file. If the tag is in a different file, then the new file will be opened for editing. If the current buffer has been modified since the last write, a warning will be written and the command will be aborted. The action can be overridden by appending the character ! to the command name. The current line indicator will be reset to the line indicated by the tag. This command is affected by the editor options **autowrite** and **writeany**; see **Edit Options in ex** on page 317 and *ctags* for details. This command is affected by the **tags** editor option.

The **tag** command will search for *tagstring* in the tag file referred to by the **tags** editor option until a reference to *tagstring* is found. The file pointed to by this reference will be loaded into the buffer, and the current line will be set to the first occurrence of the pattern specified in the tag file associated with the supplied *tagstring*; if the tag file contained a line-number reference, the current line will be set to that line. If the pattern or line number is not found, an error message will be written. If a file referred to by the **tags** editor option does not exist or is not readable, an error message will be written. The results are unspecified if the format of a tags file is not as specified by the *ctags* utility description.

Unabbreviate

Synopsis: `una[bbrev]` *word*

Delete *word* from the list of abbreviations, as described by the **abbrev** editor command.

Undo

Synopsis: `u[ndo]`

Reverse the changes made by the previous editing command (one that changes the contents of the buffer). For this purpose, **global** and **visual** are considered single commands. An **undo** can itself be reversed. Commands that affect the external environment, such as **write**, **edit** and **next,** cannot be undone.

Unmap

Synopsis: `unm[ap][!]` *x*

If no ! is specified, remove the command-mode macro definition for *x*; otherwise, remove the input-mode macro definition for *x*. See the **map** command.

Visual

Synopsis: `[`*line*`] vi[sual] [`*type*`] [`*count*`] [`*flags*`]`

Enter visual mode with the current line indicator set to *line*. The *type* is optional, and can 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 the number of lines that will initially be written; the default is the value of the editor option **window**. The command **Q** will exit visual mode. (For more information, see *vi*.) This command need not be supported on block-mode terminals.

Write

Synopsis: `[`*range*`] w[rite][!] [>>] [`*file*`]`
Synopsis: `[`*range*`] w[rite] [!] [`*file*`]`
Synopsis: `[`*range*`] wq[!] [>>] [`*file*`]`

Write the specified lines (the whole buffer, if no *range* is given) out to the file represented by the pathname *file*, writing to standard output the number of lines and bytes written.

If *file* is specified and is not the current file, and the file named by *file* exists, then the write will fail. If the current file has been changed by the **file** command and that file exists, the write will fail. In either case, the write can be forced by appending the character ! to the command name. An existing file can be appended to by appending >> to the command name. If the file does not exist, the result is implementation-dependent.

If the *file* is preceded by !, the program named in the *SHELL* environment variable will be invoked with *file* as its second argument, and the specified lines will be passed as standard input to the command. The **!** in this usage must be separated from the **w** command by at least one blank character. The special meaning of the **!** can be overridden by escaping it with a backslash character. This command is affected by the editor options **writeany** and **readonly**.

The command **wq** is equivalent to a **w** followed by a **q**; **wq!** is equivalent to **w!** followed by **q**. If the current buffer has no pathname associated with it, the **write** command will fail.

Write and Exit

Synopsis: **[***range***]** x**[**it**][**!**]** **[***file***]**

Perform a **write** command if any changes have been made to the current buffer since the last write to any file. The *range* defaults to the entire file.

Unless the command fails because an attempt to write lines to a file did not succeed, the *ex* utility will exit after an **x** command. This command is affected by the editor options **writeany** and **readonly**.

Yank

Synopsis: **[***range***]** ya**[**nk**]** **[***buffer***]** **[***count***]**

Place the specified lines in the named buffer. If no buffer is specified, the unnamed buffer will be used (where the most recently deleted or yanked text is placed by default).

Adjust Window

Synopsis: **[***line***]** z **[***type***]** **[***count***]** **[***flags***]**

If *type* is omitted, then *count* lines following the specified line (*line*) will be written. The default for *count* is the value of the editor option **window**. The *type* argument will change the position at which *line* will be written on the screen by affecting the number of lines written before and after *line*.

If *type* is specified, it will be one of the following:

– Place *line* at the bottom of the screen.

+ Place *line* at the top of the screen.

. Place *line* in the middle.

^ Write out *count* lines starting *count*∗2 lines before the addressed line; the net effect of this will be that a **z^** command following another **z** command writes the previous page.

= Centre the addressed line on the screen with a line of hyphens written immediately before and after it. The number of preceding and following lines of text written will be reduced to account for these lines of hyphens.

In all cases the current line indicator will be set to the last line written, with the exception of the = type, which causes the current line indicator to be set to that addressed in the command.

Escape

Synopsis: **!** *command*
Synopsis: **[***range***]!** *command*

Pass the remainder of the line after the **!** character to the program named in the *SHELL* environment variable for execution. A warning will be issued if the buffer has been changed since the last write. A single **!** character will be written when the command completes. The current line position will not be affected.

Within the text of *command*, **%** and **#** will be expanded as pathnames (the current and alternative pathnames, respectively), and **!** will be replaced with the text of the previous **!** command. (Thus, **!!** will repeat the previous **!** command.) If any such expansion is done, the expanded line will be echoed.

The special meanings of **%**, **#** and **!** can be overridden by escaping them with a backslash character. This command is affected by the editor options **autowrite** and **writeany**; see **Edit Options in ex** on page 317 for details.

In the second form of the **!** command, the remainder of the line after the **!** will be passed to the program named in the *SHELL* environment variable, as described above. The specified lines will be provided to the program as standard input; the resulting output will replace the specified lines.

Shift Left

Synopsis: **[***range***]** **<** **[***count***]** **[***flags***]**

Shift the specified lines to the left; the number of character positions to be shifted will be determined by the editor option **shiftwidth**. Only leading blank characters will be lost in shifting; other characters will not be affected. The current line indicator will be set to the last line changed.

Shift Right

Synopsis: **[***range***]** **>** **[***count***]** **[***flags***]**

Shift the specified lines to the right, by inserting blank characters, using tab characters where possible, as determined by the editor option **shiftwidth**. Empty lines will not be changed. The current line indicator will be set to the last line changed.

Resubstitute

Synopsis: **[***range***]** **&** **[***options***]** **[***count***]** **[***flags***]**
Synopsis: **[***range***]** **s[**ubstitute**]** **[***options***]** **[***count***]** **[***flags***]**
Synopsis: **[***range***]** **~** **[***options***]** **[***count***]** **[***flags***]**

Repeat the previous substitute command, as if & were replaced by the previous:

```
s/pattern/repl/
```

command. The same effect can obtained by omitting the:

```
/pattern/repl/
```

string in the **substitute** command. The version of the command using tilde will be the same as & and **s**, but the *pattern* used will be the last regular expression used in any command, not necessarily the one used in the last substitute command. For example, in the sequence:

```
s/red/blue/
/green
~
```

the ~ is equivalent to:

```
s/green/blue/
```

Scroll

Synopsis: *eof*

Write the next *n* lines, where *n* is the value of the editor option **scroll**. The command is invoked with the **eof** character (see the description of the *stty* **eof** character). The current line indicator will be set to the last line written. This command need not be supported on block-mode terminals.

Write Line Number

Synopsis: **[***line***] = [***flags***]**

Write the line number of the specified line (default last line). The current line position will not be affected.

Execute

Synopsis: *@ buffer*
Synopsis: ** buffer*

Execute each line of the named buffer as an *ex* command. If no buffer is specified or is specified as @ or *, the last buffer executed will be used. If there is no last buffer, an error occurs.

Regular Expressions in ex

The *ex* utility supports the basic regular expressions described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**. A null RE (/ /) is equivalent to the last RE encountered.

Regular expressions can be used in addresses to specify lines and, in some commands (for example, the **substitute** command), to specify portions of a line to be substituted.

The following constructs can be used to enhance the basic regular expressions:

\ <
: Match the beginning of a *word*. (See the definition of *word* at the beginning of **Command Descriptions in ex** on page 304.)

\ >
: Match the end of a *word*.

~
: Match the replacement part of the last **substitute** command. The tilde (~) character can be escaped in a regular expression to become a normal character with no special meaning.

When the editor option **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 ~ are treated as ordinary characters unless preceded by a \; when preceded by a \ they regain their special meaning.

Replacement Strings in ex

The character & (\& if the editor option **nomagic** is set) in the replacement string will stand for the text matched by the pattern to be replaced. The character ~ (\~ if **nomagic** is set) will be replaced by the replacement part of the previous substitute command. The sequence *n*, where *n* is an integer, will be replaced by the text matched by the pattern enclosed in the *n*th set of parentheses \(and \).

The strings \l, \u, \L and \U can be used to modify the case of elements in the replacement string (using the \& or \<*digit*>) notation. The string \l (\u) causes the first character (actually inserted by the substitution) that follows the \l (\u) to be converted to lower-case (upper-case). The strings \L (\U) causes all characters subsequent to them to be converted to lower-case

(upper-case) as they are inserted by the substitution until the string \e or \E, or the end of the replacement string, is encountered.

An example of case conversion with the **s** command is as follows:

```
:p
The cat sat on the mat.
:s/\<.at\>/\u&/gp
The Cat Sat on the Mat.
:s/S\(.*\)M/S\U\1\eM/p
The Cat SAT ON THE Mat.
```

In visual mode, a <control>-V <control>-M (or <control>-Q <control>-M) sequence in the replacement string will be mapped to a newline character, and so can be used to split lines. A literal <control>-M requires escaping by preceding it with a backslash (\^V^M or \^Q^M).

Edit Options in ex

The *ex* utility has a number of options that modify its behaviour. These options have default settings, which can be changed using the **set** command.

Options are Boolean unless otherwise specified.

autoindent, ai

[Default *off*]

If **autoindent** is set, each line in input mode will be indented (using first as many tab characters as possible, as determined by the editor option **tabstop**, and then using space characters) to align with the previous line. (Starting indentation will be determined by the line appended after, or the line inserted before or the first line changed, with an **a**, **i** or **c** command, respectively.) When a newline character is inserted in the middle of a line, and when **autoindent** is on, the first non-blank character to the right of the cursor will be aligned to the current margin on a new line immediately following the current line. Any blank characters to the left of the cursor position at which the newline character was entered will be retained. When **autoindent** is off, a new line will be created, but no blank characters will be discarded. Additional indentation can be provided as usual; succeeding lines will automatically be indented to the new alignment. A line entered during input mode with **autoindent** that contains no user-entered characters will be empty, despite the appearance of indentation during entry.

Reducing the indent can be achieved when the cursor is at the current left margin by typing <control>-D one or more times; the cursor will be moved back to the previous integral number of **shiftwidth** spaces for each <control>-D. A ^ followed by a <control>-D will remove all indentation temporarily (for the current line); a 0 followed by a <control>-D will remove all indentation permanently (for the current line and subsequent lines until input mode is reentered or until the indentation is specifically set to some other value). Changing the indent with <control>-D need not be supported on block-mode terminals.

autoprint, ap

[Default *on*]

If **autoprint** is set, the current line will be written after each command that changes buffer text. (Autoprint will be suppressed in the **global** [g and v] commands and for any command on which print operands [*flags*] are used to write explicitly the current line.)

autowrite, aw

[Default *off*]

If **autowrite** is set, when a **next**, **rewind**, **tag**, **edit**, **suspend**, **stop** or ! command is given, the buffer will be written (to the current file) if it has been modified. Appending the character ! to the command name for any of these commands except ! causes the write not to occur. If the write fails, the command will be aborted and an error message will be written.

beautify, bf

[Default *off*]

If **beautify** is set, all non-printable characters, other than tab, newline and form-feed characters, will be discarded from text read in from files.

directory, dir

EX [Default *implementation-dependent*]

The value of this option specifies the directory in which the editor buffer is to be placed. If this directory is not writable by the user, the editor quits.

edcompatible, ed

EX [Default *off*]

Causes the presence of **g** and **c** suffixes on substitute commands to be remembered, and toggled by repeating the suffixes.

errorbells, eb

[Default *off*]

If **errorbells** is set, error messages will be preceded by an alert action. Setting this option off causes the user to be informed of an error even when in visual mode, but rather than using the alert character, an error message will be written, using a standout mode of the terminal (such as inverse video) instead of the normal effect of the alert character, when the effect of the alert character is to cause the terminal to ring a bell or make other sounds. The editor should place the error message in a standout mode of the terminal, such as inverse video instead of ringing the bell, when the terminal capabilities allow this.

exrc

[Default *off*]

If **exrc** is set, *ex* will access any **.exrc** file in the current directory, as described previously. If **exrc** is not set, *ex* will ignore any **.exrc** file in the current directory during initialisation, unless the current directory is that named by the *HOME* variable.

ignorecase, ic

[Default *off*]

If **ignorecase** is set, characters that have upper-case and lower-case representations will have those representations considered as equivalent for purposes of regular expression comparison.

lisp

EX [Default *off*]

Autoindent mode and the **(**, **)**, **{**, **}**, **[[** and **]]** commands in visual mode are suitably modified for **lisp** code.

list

[Default *off*]

If **list** is set, write the addressed lines in a way that should be unambiguous: non-printable characters will be written as implementation-dependent multi-character sequences; the end of the line will be marked with a $.

magic

[Default *on*]

If **magic** is set, change the interpretation of characters in regular expressions and substitution replacement strings (see **Regular Expressions in ex** on page 316 and **Replacement Strings in ex** on page 316).

mesg

[Default *on*]

If **mesg** is set, the permission for others to use the **write** or **talk** commands to write to the terminal will be turned on while in visual mode. The shell-level command *mesg* **n** takes precedence over any setting of the *ex* **mesg** option; that is, if **mesg y** was issued before *ex* started (or in a shell escape), such as:

```
:!mesg y
```

the **mesg** option in *ex* can suppress incoming messages, but the **mesg** option cannot enable incoming messages if **mesg n** was issued.

number, nu

[Default *off*]

If **number** is set, lines will be written with line numbers, as with the **number** command.

paragraphs, para

[Default *implementation-dependent*]

The **paragraph** option defines additional paragraph boundaries for the **{** and **}** commands in visual mode. The **paragraph** option can be set to a character string consisting of zero or more character pairs. The default value is implementation-dependent.

In the text to be edited, the character string <newline>.<*char-pair*>, where <*char-pair*> is one of the character pairs found in **paragraph**, defines a paragraph boundary. For example, if:

```
paragraph=LaA ##
```

then all of the following additional paragraph boundaries would be recognised:

```
<newline>.La
<newline>.A<space>
<newline>.##
```

prompt

[Default *on*]

If **prompt** is set, command mode input will be prompted for with a colon (:); when unset, no prompt will be written.

readonly

[Default *see text*]

If **readonly** is set, read-only mode will be enabled. Writing to a different file will be allowed in read-only mode; in addition, the write can be forced by using the character ! (see the editor command **write**). The default setting will be *off* unless the file lacks write permission or the command-line option –**R** is used.

redraw

EX [Default *off*]

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

[Default *on*]

If **remap** is set, macro translation will allow for macros defined in terms of other macros; translation will continue until the final product is obtained. If unset, only a one-step translation will be done.

report

[Default 5]

The value of this option will give the number of lines that can be changed by an editor command before a report is generated on the number of lines affected.

scroll

[Default window / 2]

The value of this option will determine the number of lines scrolled on a **eof** command (see **Scroll** on page 316 and the description of the *stty* **eof** character). Changing the value of **window** in *EXINIT*, one of the **.exrc** files, or with a *set* command will not affect the value of scroll. This editor option need not be supported on block-mode terminals.

sections

[Default *implementation-dependent*]

The **sections** option defines additional section boundaries for the [[and]] commands in visual mode. The **sections** option can be set to a character string consisting of zero or more character pairs. The default value is implementation-dependent.

In the text to be edited, the character string <newline>.<*char-pair*>, where <*char-pair*> is one of the character pairs found in **sections**, defines a section boundary in the same manner that paragraph boundaries are defined. (See the **paragraphs** command.)

shell, sh

[Default from the environment *SHELL*]

The value of this option can be a string representing the pathname of the shell to be invoked for the ! shell escape command, and by the **shell** command. The default is taken from the *SHELL* variable in the environment; see **ENVIRONMENT VARIABLES** for default values for *SHELL*.

shiftwidth, sw

[Default 8]

The value of this option gives the width of an indentation level used during **autoindent** and by the shift commands.

showmatch, sm

[Default *off*]

If **showmatch** is set, in visual mode, when a) or } is typed, the matching (or { will be shown if it is still on the screen. This editor option need not be supported on block-mode terminals.

showmode

[Default *off*]

If **showmode** is set, in visual mode, the current mode that the editor is in will be written on the last line of the screen. Modes that will be reported are command mode and input mode; other unspecified modes may be written.

slowopen

EX [Default *off*]

In visual mode, this option prevents screen updates during input to improve throughput on unintelligent terminals.

tabstop, ts

[Default 8]

The value of this option specifies the software tab stops to be used by the editor to expand tabs in the input.

tags

[Default *see text*]

The value of this option can be a string representing space-character-separated pathnames that will be used as tag files for the **tag** command. A requested tag will be searched for sequentially in the specified files. By default, filenames of **tags** will be searched for in the current directory and in other implementation-dependent directories.

term

[Default from the environment *TERM*]

The value of this option can be a string representing the terminal type of the output device. The default is taken from the *TERM* variable in the environment; see **ENVIRONMENT VARIABLES** for default values for *TERM*.

terse

[Default *off*]

If **terse** is set, error messages may be less verbose. However, except for this caveat, error messages are unspecified. Furthermore, not all error messages need change for different settings of this option.

warn

[Default *on*]

If **warn** is set, *ex* will write a warning message to standard error if the contents of the buffer have not been saved before a ! command escape.

window

[Default *see text*]

The value of this option determines the default number of lines in a screenful, as written by the **z** command. When in visual mode, the number of lines output when moving up or down the file by a *screenful*. The value of **window** can be unrelated to the real screen size, except that it will be set on entry to be the current number of screen lines. (The current number of screen lines will be determined by the system or overridden by the user, as described for *LINES* in **ENVIRONMENT VARIABLES** and the **XBD** specification, **Chapter 6**, **Environment Variables**.) The baud rate of the terminal line may reduce the default in an implementation-dependent manner. The default value of **windows** also can be overridden by specifying a window size using the **–w** command-line option.

wrapscan, ws

[Default *on*]

If **wrapscan** is set, searches (using / / or ??) will wrap around the end of the editing buffer; when unset, searches will stop at the beginning of the editing buffer for ??, or at the end of the editing buffer for / /.

wrapmargin, wm

[Default 0]

If the value of this option is greater than zero (say *n*) in visual mode during text entry, then, in the POSIX locale, a newline character will replace all consecutive blank characters, at the boundary between a blank character and a non-blank character, so that lines will end at least *n* spaces from the ending margin of the terminal screen. (The ending margin will be determined by the system or overridden by the user, as described for *COLUMNS* in **ENVIRONMENT VARIABLES** and the **XBD** specification, **Chapter 6**, **Environment Variables**.) If a line consists of a sequence of non-blank characters long enough such that it extends continuously from the beginning margin to beyond the ending margin, that sequence will not be broken by the action of this option. If the value is zero, no wrapping will be performed.

writeany, wa

[Default *off*]

If **writeany** is set, file-overwriting checks will be inhibited that would otherwise be made before **write** and **xit** commands, or before an automatic write (see editor option **autowrite**), allowing a write to any file (provided permissions allow it).

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When an error in the input script is encountered, or when an error is detected that is a consequence of the data (not) present in the file or due to an external condition such as a read or write error:

- If the standard input is a terminal device file, all input will be flushed, and a new command read.
- If the standard input is a regular file, *ex* will terminate with a non-zero exit status.
- If the command in error was one of those specified with the command-line **–c** option (or the obsolescent + option), all of the remaining **–c** commands will be flushed, and a new command read from standard input.

APPLICATION USAGE

If a SIGSEGV signal is received while *ex* is saving a file, the file might not be successfully saved.

The **next** command can accept more than one file, so usage such as:

```
next 'ls [abc]*'
```

is valid; it would not be valid for the **edit** or **read** commands, for example, because they expect only one file and unspecified results occur.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *sed*, *vi*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The ~ regular expression notation has been correctly documented.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

expand – convert tabs to spaces

SYNOPSIS

```
expad [-t tablist][file ...]
```

OB

```
expand [-tabstop][-tab1,tab2,...,tabn][file...]
```

DESCRIPTION

The *expand* utility writes files or the standard input to the standard output with tab characters replaced with one or more space characters needed to pad to the next tab stop. Any backspace characters will be copied to the output and cause the column position count for tab stop calculations to be decremented; the column position count will not be decremented below zero.

OPTIONS

OB The *expand* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**; the obsolescent version does not.

The following options are supported:

–t *tablist*

Specify the tab stops. The argument *tablist* must consist of a single positive decimal integer or multiple positive decimal integers, separated by blank characters or commas, in ascending order. If a single number is given, tabs will be set *tablist* column positions apart instead of the default 8. If multiple numbers are given, the tabs will be set at those specific column positions.

Each tab-stop position *N* must be an integer value greater than zero, and the list must be in strictly ascending order. This is taken to mean that, from the start of a line of output, tabbing to position *N* causes the next character output to be in the (*N*+1)th column position on that line.

In the event of *expand* having to process a tab character at a position beyond the last of those specified in a multiple tab-stop list, the tab character is replaced by a single space character in the output.

OB In the obsolescent version, the single number is specified as *tabstop* with a leading minus; multiple tab stops are specified after a leading minus as *tab1*, *tab2* and so forth.

OPERANDS

The following operand is supported:

file The pathname of a text file to be used as input.

STDIN

See **INPUT FILES**.

INPUT FILES

Input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *expand*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the processing of tab and space characters, and for the determination of the width in column positions each character would occupy on a constant-width font output device.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output is equivalent to the input files with tab characters converted into the appropriate number of space characters.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion
>0 An error occurred.

CONSEQUENCES OF ERRORS
The *expand* utility will terminate with an error message and non-zero exit status upon encountering difficulties accessing one of the *file* operands.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
tabs, unexpand.

CHANGE HISTORY
First released in Issue 4.

NAME

expr – evaluate arguments as an expression

SYNOPSIS

```
expr operand
```

DESCRIPTION

The *expr* utility will evaluate an expression and write the result to standard output.

OPTIONS

None.

OPERANDS

The single expression evaluated by *expr* will be formed from the operands, as described in **EXTENDED DESCRIPTION**. Each of the expression operator symbols:

```
(   )   |   &   =   >   >=   <   <=   !=   +   –   *   /   %   :
```

and the symbols *integer* and *string* in the table must be provided as separate arguments to *expr*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *expr*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions and by the string comparison operators.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments) and the behaviour of character classes within regular expressions.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *expr* utility will evaluate the expression and write the result to standard output. The character 0 will be written to indicate a zero value and nothing will be written to indicate a null string.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The formation of the expression to be evaluated is shown in the following table. The symbols *expr*, *expr1* and *expr2* represent expressions formed from *integer* and *string* symbols and the expression operator symbols (all separate arguments) by recursive application of the constructs described in the table. The expressions are listed in order of increasing precedence, with equal-precedence operators grouped between horizontal lines. All of the operators are left-associative.

Expression	Description
expr1 **|** *expr2*	Returns the evaluation of *expr1* if it is neither null nor zero; otherwise, returns the evaluation of *expr2*.
expr1 **&** *expr2*	Returns the evaluation of *expr1* if neither expression evaluates to null or zero; otherwise, returns zero.
	Returns the result of a decimal integer comparison if both arguments are integers; otherwise, returns the result of a string comparison using the locale-specific collation sequence. The result of each comparison will be 1 if the specified relationship is true, or 0 if the relationship is false.
expr1 **=** *expr2*	Equal.
expr1 **>** *expr2*	Greater than.
expr1 **>=** *expr2*	Greater than or equal.
expr1 **<** *expr2*	Less than.
expr1 **<=** *expr2*	Less than or equal.
expr1 **!=** *expr2*	Not equal.
expr1 **+** *expr2*	Addition of decimal integer-valued arguments.
expr1 **–** *expr2*	Subtraction of decimal integer-valued arguments.
expr1 ***** *expr2*	Multiplication of decimal integer-valued arguments.
expr1 **/** *expr2*	Integer division of decimal integer-valued arguments, producing an integer result.
expr1 **%** *expr2*	Remainder of integer division of decimal integer-valued arguments.
expr1 **:** *expr2*	Matching expression. See below.
(expr **)**	Grouping symbols. Any expression can be placed within parentheses. Parentheses can be nested to a depth of {EXPR_NEST_MAX}.
integer	An argument consisting only of an (optional) unary minus followed by digits.
string	A string argument. See below.

Matching Expression

The : matching operator will compare the string resulting from the evaluation of *expr1* with the regular expression pattern resulting from the evaluation of *expr2*. Regular expression syntax is that defined in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**, except that all patterns are anchored to the beginning of the string (that is, only sequences starting at the first character of a string will be matched by the regular expression) and, therefore, it is unspecified whether ^ is a special character in that context. Usually, the matching operator will return a string representing the number of characters matched ("0" on failure). Alternatively, if the pattern contains at least one regular expression subexpression [**\(...**\)], the string corresponding to \1 will be returned.

String Operand

A string argument is an argument that cannot be identified as an *integer* argument or as one of the expression operator symbols shown in **OPERANDS**.

The use of string arguments **length**, **substr**, **index** or **match** produces unspecified results.

EXIT STATUS

The following exit values are returned:

0 The *expression* evaluates to neither null nor zero.
1 The *expression* evaluates to null or zero.
2 Invalid *expression*.
>2 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

After argument processing by the shell, *expr* is not required to be able to 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 all may be taken as the = operator). The following works reliably:

```
expr X$a = X=
```

Also note that this document permits implementations to extend utilities. The *expr* utility permits the integer arguments to be preceded with a unary minus. This means that an integer argument could look like an option. Therefore, the portable application must employ the −− construct of Guideline 10 of the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** to protect its operands if there is any chance the first operand might be a negative integer (or any string with a leading minus).

EXAMPLES

The *expr* utility has a rather difficult syntax:

- Many of the operators are also shell control operators or reserved words, so they have to be escaped on the command line.

- Each part of the expression is composed of separate arguments, so liberal usage of blank characters is required. For example:

Invalid	Valid
`expr 1+2`	`expr 1 + 2`
`expr "1 + 2"`	`expr 1 + 2`
`expr 1 + (2 * 3)`	`expr 1 + \( 2 \* 3 \)`

In many cases, the arithmetic and string features provided as part of the shell command language are easier to use than their equivalents in *expr*. Newly written scripts should avoid *expr* in favour of the new features within the shell. See Section 2.5 on page 27 and Section 2.6.4 on page 38.

The following command:

```
a=$(expr $a + 1)
```

adds 1 to the variable *a*.

The following command, for **$a** equal to either **/usr/abc/file** or just **file:**

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

returns the last segment of a pathname (that is, **file**). Applications should avoid the character / used alone as an argument: *expr* may interpret it as the division operator.

The following command:

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

is a better representation of the previous example. The addition of the // characters eliminates any ambiguity about the division operator and simplifies the whole expression. Also note that pathnames may contain characters contained in the *IFS* variable and should be quoted to avoid having **$a** expand into multiple arguments.

The following command:

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

returns the number of characters in *VAR*.

FUTURE DIRECTIONS

None.

SEE ALSO

Section 2.6.4 on page 38.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

false – return false value

SYNOPSIS

```
false
```

DESCRIPTION

The *false* utility will return with a non-zero exit code.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The *false* utility always will exit with a value other than zero.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

true.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

fc – process command history list

SYNOPSIS

```
fc [-r][-e editor] [first[last]]
fc -l[-nr] [first[last]]
fc -s[old=new][first]
```

DESCRIPTION

The *fc* utility lists or edits and reexecutes, commands previously entered to an interactive *sh*.

The command history list references commands by number. The first number in the list is selected arbitrarily. The relationship of a number to its command will not change except when the user logs in and no other process is accessing the list, at which time the system may reset the numbering to start the oldest retained command at another number (usually 1). When the number reaches an implementation-dependent upper limit, which will be no smaller than the value in *HISTSIZE* or 32 767 (whichever is greater), the shell may wrap the numbers, starting the next command with a lower number (usually 1). However, despite this optional wrapping of numbers, *fc* will maintain the time-ordering sequence of the commands. For example, if four commands in sequence are given the numbers 32 766, 32 767, 1 (wrapped), and 2 as they are executed, command 32 767 is considered the command previous to 1, even though its number is higher.

When commands are edited (when the **–l** option is not specified), the resulting lines will be entered at the end of the history list and then reexecuted by *sh*. The *fc* command that caused the editing will not be entered into the history list. If the editor returns a non-zero exit status, this will suppress the entry into the history list and the command reexecution. Any command-line variable assignments or redirection operators used with *fc* will affect both the *fc* command itself as well as the command that results, for example:

```
fc -s -- -1 2>/dev/null
```

reinvokes the previous command, suppressing standard error for both *fc* and the previous command.

OPTIONS

The *fc* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–e *editor*
: Use the editor named by *editor* to edit the commands. The *editor* string is a utility name, subject to search via the *PATH* variable (see the **XBD** specification, **Chapter 6**, **Environment Variables**). The value in the *FCEDIT* variable is used as a default when **–e** is not specified. If *FCEDIT* is null or unset, *ed* will be used as the editor.

–l
: (The letter ell.) List the commands rather than invoking an editor on them. The commands will be written in the sequence indicated by the *first* and *last* operands, as affected by **–r**, with each command preceded by the command number.

–n
: Suppress command numbers when listing with **–l**.

–r
: Reverse the order of the commands listed (with **–l**) or edited (with neither **–l** nor **–s**).

–s Reexecute the command without invoking an editor.

OPERANDS

The following operands are supported:

first
last Select the commands to list or edit. The number of previous commands that can be accessed is determined by the value of the *HISTSIZE* variable. The value of *first* or *last* or both will be one of the following:

[+]*number*
A positive number representing a command number; command numbers can be displayed with the **–l** option.

–number
A negative decimal number representing the command that was executed *number* of commands previously. For example, –1 is the immediately previous command.

string A string indicating the most recently entered command that begins with that string. If the *old=new* operand is not also specified with **–s**, the string form of the *first* operand cannot contain an embedded equal sign.

When the synopsis form with **–s** is used:

- If *first* is omitted, the previous command will be used.

For the synopsis forms without **–s**:

- If *last* is omitted, *last* defaults to the previous command when **–l** is specified; otherwise, it defaults to *first*.
- If *first* and *last* are both omitted, the previous 16 commands will be listed or the previous single command will be edited (based on the **–l** option).
- If *first* and *last* are both present, all of the commands from *first* to *last* will be edited (without **–l**) or listed (with **–l**). Editing multiple commands will be accomplished by presenting to the editor all of the commands at one time, each command starting on a new line. If *first* represents a newer command than *last*, the commands will be listed or edited in reverse sequence, equivalent to using **–r**. For example, the following commands on the first line are equivalent to the corresponding commands on the second:

```
fc -r 10 20          fc    30 40
fc    20 10          fc -r 40 30
```

- When a range of commands is used, it will not be an error to specify *first* or *last* values that are not in the history list; *fc* will substitute the value representing the oldest or newest command in the list, as appropriate. For example, if there are only ten commands in the history list, numbered 1 to 10:

```
fc -l
fc 1 99
```

will list and edit, respectively, all ten commands.

old=new
Replace the first occurrence of string *old* in the commands to be reexecuted by the string *new*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *fc*:

FCEDIT

This variable, when expanded by the shell, determines the default value for the **−e** *editor* option's *editor* option-argument. If *FCEDIT* is null or unset, *ed* will be used as the editor.

HISTFILE

Determine a pathname naming a command history file. If the *HISTFILE* variable is not set, the shell may attempt to access or create a file **.sh_history** in the user's home directory. If the shell cannot obtain both read and write access to, or create, the history file, it will use an unspecified mechanism that allows the history to operate properly. (References to history ''file'' in this section are understood to mean this unspecified mechanism in such cases.) An implementation may choose to access this variable only when initialising the history file; this initialisation will occur when *fc* or *sh* first attempt to retrieve entries from, or add entries to, the file, as the result of commands issued by the user, the file named by the *ENV* variable, or implementation-dependent system startup files. (The initialisation process for the history file can be dependent on the system startup files, in that they may contain commands that will effectively preempt the user's settings of *HISTFILE* and *HISTSIZE*. For example, function definition commands are recorded in the history file, unless the *set* **−o nolog** option is set. If the system administrator includes function definitions in some system startup file called before the *ENV* file, the history file will be initialised before the user gets a chance to influence its characteristics.) In some historical shells, the history file is initialised just after the *ENV* file has been processed. Therefore, it is implementation-dependent whether changes made to *HISTFILE* after the history file has been initialised are effective. Implementations may choose to disable the history list mechanism for users with appropriate privileges who do not set *HISTFILE*; the specific circumstances under which this will occur are implementation-dependent. If more than one instance of the shell is using the same history file, it is unspecified how updates to the history file from those shells interact. As entries are deleted from the history file, they will be deleted oldest first. It is unspecified when history file entries are physically removed from the history file.

HISTSIZE

Determine a decimal number representing the limit to the number of previous commands that are accessible. If this variable is unset, an unspecified default greater than or equal to 128 will be used. The maximum number of commands in the history list is unspecified, but will be at least 128. An implementation may choose to access this variable only when initialising the history file, as described under *HISTFILE*. Therefore, it is unspecified whether changes made to *HISTSIZE* after the history file has been initialised are effective.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When the **–l** option is used to list commands, the format of each command in the list is as follows:

```
"%d\t%s\n", <line number>, <command>
```

If both the **–l** and **–n** options are specified, the format of each command is:

```
"\t%s\n", <command>
```

If the <*command*> consists of more than one line, the lines after the first are displayed as:

```
"\t%s\n", <continued-command>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion of the listing.
>0 An error occurred.

Otherwise, the exit status will be that of the commands executed by *fc*.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Since editors sometimes use file descriptors as integral parts of their editing, redirecting their file descriptors as part of the *fc* command can produce unexpected results. For example, if *vi* is the *FCEDIT* editor, the command:

```
fc -s | more
```

will not work correctly on many systems.

Users on windowing systems may want to have separate history files for each window by setting *HISTFILE* as follows:

```
HISTFILE=$HOME/.sh_hist$$
```

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sh.

CHANGE HISTORY

First released in Issue 4.

NAME

fg – run jobs in the foreground

SYNOPSIS

JC `fg [`*job_id*`]`

DESCRIPTION

If job control is enabled (see the description of *set* –**m**), the *fg* utility will move a background job from the current environment (see Section 2.12 on page 63) into the foreground.

Using *fg* to place a job into the foreground will remove its process ID from the list of those ‘‘known in the current shell execution environment’’; see Section 2.9.3 on page 50.

OPTIONS

None.

OPERANDS

The following operand is supported:

job_id Specify the job to be run as a foreground job. If no *job_id* operand is given, the *job_id* for the job that was most recently suspended, placed in the background or run as a background job will be used. The format of *job_id* is described in the entry for **job control job ID** in the **XBD** specification, **Chapter 2**, **Glossary**.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *fg*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *fg* utility writes the command line of the job to standard output in the following format:

```
"%s\n", <command>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If job control is disabled, the *fg* utility will exit with an error and no job will be placed in the foreground.

APPLICATION USAGE

The *fg* utility will not work as expected when it is operating in its own utility execution environment because that environment will have no applicable jobs to manipulate. See the **APPLICATION USAGE** section for *bg*. For this reason, *fg* is generally implemented as a shell regular built-in.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

bg, *kill*, *jobs*, *wait*.

CHANGE HISTORY

First released in Issue 4.

NAME

fgrep – search a file for a fixed-string pattern

SYNOPSIS

OB `fgrep [ -c| -l][-invx] -e pattern_list [file...]`

OB `fgrep [ -c| -l][-invx] -f pattern_list [file...]`

OB `fgrep [ -c| -l][-invx] pattern_list [file...]`

DESCRIPTION

The name *fgrep* is an obsolescent version equivalent to *grep* **–F**.

A command invoking the *fgrep* utility with the **–e** option specified is equivalent to the command:

```
grep -E [ -c| -l][-invx] -e pattern_list [file...]
```

A command invoking the *fgrep* utility with the **–f** option specified is equivalent to the command:

```
grep -E [ -c| -l][-invx] -f pattern_list [file...]
```

A command invoking the *fgrep* utility with neither the **–e** nor the **–f** option operand specified is equivalent to the command:

```
grep -E [ -c| -l][-invx] pattern_list [file...]
```

APPLICATION USAGE

Unlike *grep* **–F**, multiple **–e** or **–f** options produce undefined results. Adjacent newline characters in the *pattern* operand or **–e** *pattern_list* option-argument also produce undefined results.

FUTURE DIRECTIONS

The *fgrep* utility may be withdrawn from a future issue; applications should migrate to *grep* **–F**.

SEE ALSO

grep.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Separated from the *egrep* description.

NAME

file – determine file type

SYNOPSIS

`file` *`file ...`*

DESCRIPTION

The *file* utility performs a series of tests on each specified *file* in an attempt to classify it:

1. If the file is not a regular file, its file type is identified. The file types directory, FIFO, block special and character special are identified as such. Other implementation-dependent file types may also be identified.
2. If the file is a regular file, and:
   a. The file is zero-length, it is identified as an empty file.
   b. The file is not zero-length, *file* will examine an initial segment of the file and make a guess at identifying its contents or whether it is an executable binary file. (The answer is not guaranteed to be correct.)

If *file* does not exist, cannot be read, or its file status could not be determined, the output will indicate that the file was processed, but that its type could not be determined.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of a file to be tested.

STDIN

Not used.

INPUT FILES

The *file* can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *file*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
In the POSIX locale, the following format is used to identify each operand, *file* specified:

```
"%s: %s\n", <file>, <type>
```

The values for <*type*> are unspecified, except that in the POSIX locale, if *file* is identified as one of the types listed in the following table, <*type*> will contain (but is not limited to) the corresponding string. Each space shown in the strings is exactly one space character.

If *file* is a	**<*type*> contains the string**
directory	directory
FIFO	fifo
block special	block special
character special	character special
executable binary	executable
empty regular file	empty
ar archive library (see *ar*)	archive
extended *cpio* format (see *pax*)	cpio archive
extended *tar* format (see *ustar* — see 2pax)	tar archive
shell script	commands text
C-language source	c program text
FORTRAN source	fortran program text

Table 3-6 File Utility Output Strings

If the file named by the *file* operand does not exist, cannot be read or the status of the file cannot be determined, the string **cannot open** will be included as part of the <*type*> field, but this is not considered an error that affects the exit status.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE

The *file* utility can only be required to guess at many of the file types because only exhaustive testing can determine some types with certitude. For example, binary data on some systems might match the initial segment of an executable or a *tar* archive.

Note that the table indicates that the output contains the stated string. Systems may add text before or after the string. For executables, as an example, the machine architecture and various facts about how the file was link-edited may be included.

EXAMPLES

Determine if an argument is a binary executable file:

```
file "$1" | grep -Fq executable &&
       printf "%s is executable.\n" "$1"
```

FUTURE DIRECTIONS

None.

SEE ALSO

ls.

CHANGE HISTORY

First released in Issue 4.

NAME

find – find files

SYNOPSIS

`find` *`path...`* **[***`operand_expression`***]**

DESCRIPTION

The *find* utility will recursively descend the directory hierarchy from each file specified by *path*, evaluating a Boolean expression composed of the primaries described in **OPERANDS** for each file encountered.

The *find* utility will be able to descend to arbitrary depths in a file hierarchy and will not fail due to path length limitations (unless a *path* operand specified by the application exceeds {PATH_MAX} requirements).

OPTIONS

None.

OPERANDS

The following operands are supported:

The *path* operand is a pathname of a starting point in the directory hierarchy.

The first argument that starts with a –, or is a ! or a (, and all subsequent arguments will be interpreted as an *expression* made up of the following primaries and operators. In the descriptions, wherever *n* is used as a primary argument, it will be interpreted as a decimal integer optionally preceded by a plus (+) or minus (–) sign, as follows:

+*n* more than *n*

n exactly *n*

–*n* less than *n*.

The following primaries are supported:

–name *pattern*
The primary will evaluate as true if the basename of the filename being examined matches *pattern* using the pattern matching notation described in Section 2.13 on page 64.

–nouser The primary will evaluate as true if the file belongs to a user ID for which the **XSH** specification *getpwuid*( ) (or equivalent) function returns NULL.

–nogroup
The primary will evaluate as true if the file belongs to a group ID for which the **XSH** specification *getgrgid*( ) (or equivalent) function returns NULL.

–xdev The primary always will evaluate as true; it will cause *find* not to continue descending past directories that have a different device ID (*st_dev*, see the **XSH** specification *stat*( ) function). If any **–xdev** primary is specified, it will apply to the entire expression even if the **–xdev** primary would not normally be evaluated.

–prune The primary always will evaluate as true; it will cause *find* not to descend the current pathname if it is a directory. If the **–depth** primary is specified, the **–prune** primary will have no effect.

–perm [–]*mode*
The *mode* argument is used to represent file mode bits. It will be identical in format to the *symbolic_mode* operand described in *chmod* on page 185, and will be interpreted as follows. To start, a template will be assumed with all file mode bits cleared. An *op*

symbol of + will set the appropriate mode bits in the template; – will clear the appropriate bits; = will set the appropriate mode bits, without regard to the contents of process' file mode creation mask. The *op* symbol of – cannot be the first character of *mode*; this avoids ambiguity with the optional leading hyphen. Since the initial mode is all bits off, there are not any symbolic modes that need to use – as the first character.

If the hyphen is omitted, the primary will evaluate as true when the file permission bits exactly match the value of the resulting template.

Otherwise, if *mode* is prefixed by a hyphen, the primary will evaluate as true if at least all the bits in the resulting template are set in the file permission bits.

EX **–perm [–]***onum*
If the hyphen is omitted, the primary will evaluate as true when the file permission bits exactly match the value of the octal number *onum* and only the bits corresponding to the octal mask 07777 will be compared. (See the description of the octal *mode* in *chmod*.) Otherwise, if *onum* is prefixed by a hyphen, the primary will evaluate as true if at least all of the bits specified in *onum* that are also set in the octal mask 07777 are set.

–type *c* The primary will evaluate as 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 or regular file, respectively.

–links *n*
The primary will evaluate as true if the file has *n* links.

–user *uname*
The primary will evaluate as true if the file belongs to the user *uname*. If *uname* is a decimal integer and the *getpwnam*( ) (or equivalent) function does not return a valid user name, *uname* will be interpreted as a user ID.

–group *gname*
The primary will evaluate as true if the file belongs to the group *gname*. If *gname* is a decimal integer and the *getgrnam*( ) (or equivalent) function does not return a valid group name, *gname* will be interpreted as a group ID.

–size *n***[c]**
The primary will evaluate as true if the file size in bytes, divided by 512 and rounded up to the next integer, is *n*. If *n* is followed by the character c, the size will be in bytes.

–atime *n*
The primary will evaluate as true if the file access time subtracted from the initialisation time is *n*–1 to *n* multiples of 24 hours. The initialisation time will be a time between the invocation of the *find* utility and the first access by that invocation of the *find* utility to any file specified by its *path* operands. For example, **–atime** 3 is true if the file was accessed any time in the period from 72 to 48 hours ago.

–mtime *n*
The primary will evaluate as true if the file modification time subtracted from the initialisation time is *n*–1 to *n* multiples of 24 hours. The initialisation time will be a time between the invocation of the *find* utility and the first access by that invocation of the *find* utility to any file specified by its *path* operands.

–ctime *n*
The primary will evaluate as true if the time of last change of file status information subtracted from the initialisation time is *n*–1 to *n* multiples of 24 hours. The initialisation time will be a time between the invocation of the *find* utility and the first access by that invocation of the *find* utility to any file specified by its *path* operands.

–**exec** *utility_name* **[***argument* ...**]** **;**
The primary will evaluate as true if the invoked utility *utility_name* returns a zero value as exit status. The end of the primary expression will be punctuated by a semicolon. A *utility_name* or *argument* containing only the two characters {} will be replaced by the current pathname. If a *utility_name* or argument string contains the two characters {}, but not just the two characters {}, it is implementation-dependent whether *find* replaces those two characters with the current pathname or uses the string without change. The current directory for the invocation of *utility_name* will be the same as the current directory when the *find* utility was started. If the *utility_name* names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

–**ok** *utility_name* **[***argument* ...**]** **;**
The –**ok** primary will be equivalent to –**exec**, except that *find* will request affirmation of the invocation of *utility_name* using the current file as an argument by writing to standard error as described in **STDERR**. If the response on standard input is affirmative, the utility will be invoked. Otherwise, the command will not be invoked and the value of the –**ok** operand will be false.

–**print** The primary always will evaluate as true; it will cause the current pathname to be written to standard output.

–**newer** *file*
The primary will evaluate as true if the modification time of the current file is more recent than the modification time of the file named by the pathname *file*.

–**depth** The primary always will evaluate as true; it will cause descent of the directory hierarchy to be done so that all entries in a directory are acted on before the directory itself. If a –**depth** primary is not specified, all entries in a directory will be acted on after the directory itself. If any –**depth** primary is specified, it will apply to the entire expression even if the –**depth** primary would not normally be evaluated.

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

(*expression*)
True if *expression* is true.

! *expression*
Negation of a primary; the unary NOT operator.

expression **[**–**a]** *expression*
Conjunction of primaries; the AND operator will be implied by the juxtaposition of two primaries or made explicit by the optional –**a** operator. The second expression will not be evaluated if the first expression is false.

expression –**o** *expression*
Alternation of primaries; the OR operator. The second expression will not be evaluated if the first expression is true.

If no *expression* is present, –**print** will be used as the expression. Otherwise, if the given expression does not contain any of the primaries –**exec**, –**ok** or –**print**, the given expression will be effectively replaced by:

```
( given_expression ) -print
```

The –**user**, –**group** and –**newer** primaries each will evaluate their respective arguments only once.

STDIN

If the −**ok** primary is used, the response will be read from the standard input. An entire line will be read as the response. Otherwise, the standard input will not be used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *find*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the pattern matching notation for the −**n** option and in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE

This variable will determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments), the behaviour of character classes within the pattern matching notation used for the −**n** option, and the behaviour of character classes within regular expressions used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES

Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the location of the *utility_name* for the −**exec** and −**ok** primaries, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The −**print** primary will cause the current pathnames to be written to standard output. The format will be:

```
"%s\n", <path>
```

STDERR

The −**ok** primary will write a prompt to standard error containing at least the *utility_name* to be invoked and the current pathname. In the POSIX locale, the last non-blank character in the prompt will be ?. The exact format used is unspecified.

Otherwise, the standard error will be used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All *path* operands were traversed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

When used in operands, pattern matching notation, semicolons, opening parentheses, and closing parentheses are special to the shell and must be quoted (see Section 2.2 on page 20).

The bit that is traditionally used for sticky (historically 01000) is still specified in the –**perm** primary using the octal number argument form. Since this bit is not defined by this document, applications must not assume that it actually refers to the traditional sticky bit.

The references to octal modes are marked EX because, although they are obsolescent in the ISO/IEC 9945-2:1993 standard, X/Open is committed to maintaining them for portable applications until further notice.

EXAMPLES

1. The following commands are equivalent:

```
find .
find . -print
```

   They both write out the entire directory hierarchy from the current directory.

2. The following command:

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

   removes all files named **tmp** or ending in .xx that have not been accessed for seven or more 24-hour periods.

3. The following command:

```
find . -perm -o+w,+s
```

   prints (–**print** is assumed) the names of all files in or below the current directory, with all of the file permission bits S_ISUID, S_ISGID and S_IWOTH set.

4. The following command:

```
find . -name SCCS -prune -o -print
```

   recursively prints pathnames of all files in the current directory and below, but skips directories named SCCS and files in them.

5. The following command:

```
find . -print -name SCCS -prune
```

   behaves as in the previous example, but prints the names of the SCCS directories.

6. The following command is roughly equivalent to the –**nt** extension to *test*:

```
if [ -n "$(find file1 -prune -newer file2)" ]; then
       printf %s\\n "file1 is newer than file2"
fi
```

7. The descriptions of –**atime**, –**ctime** and –**mtime** use the terminology *n* ‘‘24-hour periods’’. For example, a file accessed at 23:59 will be selected by:

```
find . -atime -1 -print
```

   at 00:01 the next day (less than 24 hours later, not more than one day ago); the midnight boundary between days has no effect on the 24-hour calculation.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, *pax*, *sh*, *test*, the **XSH** specification description of *stat*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

fold – filter for folding lines

SYNOPSIS

`fold [-bs][-w` *`width`*`][`*`file...`*`]`

DESCRIPTION

The *fold* utility is a filter that will fold lines from its input files, breaking the lines to have a maximum of *width* column positions (or bytes, if the –**b** option is specified). Lines will be broken by the insertion of a newline character such that each output line (referred to later in this section as a segment) is the maximum width possible that does not exceed the specified number of column positions (or bytes). A line will not be broken in the middle of a character. The behaviour is undefined if *width* is less than the number of columns any single character in the input would occupy.

If the carriage-return, backspace or tab characters are encountered in the input, and the –**b** option is not specified, they will be treated specially:

backspace
: The current count of line width will be decremented by one, although the count never will become negative. The *fold* utility will not insert a newline character immediately before or after any backspace character.

carriage-return
: The current count of line width will be set to zero. The *fold* utility will not insert a newline character immediately before or after any carriage-return character.

tab
: Each tab character encountered will advance the column position pointer to the next tab stop. Tab stops will be at each column position *n* such that *n* modulo 8 equals 1.

OPTIONS

The *fold* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**b**
: Count *width* in bytes rather than column positions.

–**s**
: If a segment of a line contains a blank character within the first *width* column positions (or bytes), break the line after the last such blank character meeting the width constraints. If there is no blank character meeting the requirements, the –**s** option will have no effect for that output segment of the input line.

–**w** *width*
: Specify the maximum line length, in column positions (or bytes if –**b** is specified). The results are unspecified if *width* is not a positive decimal number. The default value is 80.

OPERANDS

The following operand is supported:

file
: A pathname of a text file to be folded. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

If the –**b** option is specified, the input files must be text files except that the lines are not limited to {LINE_MAX} bytes in length. If the –**b** option is not specified, the input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *fold*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), and for the determination of the width in column positions each character would occupy on a constant-width font output device.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output will be a file containing a sequence of characters whose order will be preserved from the input files, possibly with inserted newline characters.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All input files were processed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *cut* and *fold* utilities can be used to create text files out of files with arbitrary line lengths. The *cut* utility should be used when the number of lines (or records) needs to remain constant. The *fold* utility should be used when the contents of long lines need to be kept contiguous.

The *fold* utility is frequently used to send text files to line printers that truncate, rather than fold, lines wider than the printer is able to print (usually 80 or 132 column positions).

EXAMPLES

An example invocation that submits a file of possibly long lines to the line printer (under the assumption that the user knows the line width of the printer to be assigned by *lp*):

```
fold -w 132 bigfile | lp
```

FUTURE DIRECTIONS

None.

SEE ALSO

cut.

CHANGE HISTORY

First released in Issue 4.

NAME

fort77 – FORTRAN compiler (**FORTRAN**)

SYNOPSIS

`fort77 [-c][-g][-L` *directory*`]... [-O` *optlevel*`][-o` *outfile*`][-s][-w]` *operand*`...`

DESCRIPTION

The *fort77* utility is the interface to the FORTRAN compilation system; it will accept the full FORTRAN-77 language defined by the ANSI X3.9-1978 standard. The system conceptually consists of a compiler and link editor. The files referenced by *operand*s are compiled and linked to produce an executable file. It is unspecified whether the linking occurs entirely within the operation of *fort77*; some systems may produce objects that are not fully resolved until the file is executed.

If the **–c** option is present, for all pathname operands of the form *file*.f, the files:

`$(basename` *pathname* `.f).o`

will be created or overwritten as the result of successful compilation. If the **–c** option is not specified, it is unspecified whether such .o files are created or deleted for the *file*.f operands.

If there are no options that prevent link editing (such as **–c**) and all operands compile and link without error, the resulting executable file will be written into the file named by the **–o** option (if present) or to the file **a.out**. The executable file will be created as specified in the **XSH** specification, except that the file permissions will be set to:

S_IRWXO | S_IRWXG | S_IRWXU

and that the bits specified by the *umask* of the process will be cleared.

OPTIONS

The *fort77* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that:

- The **–l** *library* operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.
- The order of specifying the multiple **–L** options is significant.
- Portable applications must specify each option separately; that is, grouping option letters (for example, **–cg**) need not be recognised by all implementations.

The following options are supported:

–c Suppress the link-edit phase of the compilation, and do not remove any object files that are produced.

–g Produce symbolic information in the object or executable files; the nature of this information is unspecified, and may be modified by implementation-dependent interactions with other options.

–s Produce object or executable files, or both, from which symbolic and other information not required for proper execution using the **XSH** specification *exec* family has been removed (stripped). If both **–g** and **–s** options are present, the action taken is unspecified.

–o *outfile*
Use the pathname *outfile*, instead of the default **a.out**, for the executable file produced. If the **–o** option is present with **–c**, the result is unspecified.

–L *directory*
Change the algorithm of searching for the libraries named in **–l** operands to look in the directory named by the *directory* pathname before looking in the usual places. Directories named in **–L** options will be searched in the specified order. At least ten instances of this option will be supported in a single *fort77* command invocation. If a directory specified by a **–L** option contains a file named **libf.a**, the results are unspecified.

–O *optlevel*
Specify the level of code optimisation. If the *optlevel* option-argument is the digit 0, all special code optimisations will be disabled. If it is the digit 1, the nature of the optimisation is unspecified. If the **–O** option is omitted, the nature of the system's default optimisation is unspecified. It is unspecified whether code generated in the presence of the **–O 0** option is the same as that generated when **–O** is omitted. Other *optlevel* values may be supported.

–w Suppress warnings.

Multiple instances of **–L** options can be specified.

OPERANDS

An *operand* is either in the form of a pathname or the form **–l** *library*. At least one operand of the pathname form will be specified. The following operands are supported:

file.**f** The pathname of a FORTRAN source file to be compiled and optionally passed to the link editor. The filename operand will be of this form if the **–c** option is used.

file.**a** A library of object files typically produced by *ar*, and passed directly to the link editor. Implementations may recognise implementation-dependent suffixes other than **.a** as denoting object file libraries.

file.**o** An object file produced by *fort77* **–c** and passed directly to the link editor. Implementations may recognise implementation-dependent suffixes other than **.o** as denoting object files.

The processing of other files is implementation-dependent.

–l *library*
(The letter ell.) Search the library named:

`lib`*`library`*`.a`

A library is searched when its name is encountered, so the placement of a **–l** operand is significant. Several standard libraries can be specified in this manner, as described in **EXTENDED DESCRIPTION**. Implementations may recognise implementation-dependent suffixes other than **.a** as denoting libraries.

STDIN

Not used.

INPUT FILES

The input file must be one of the following: a text file containing FORTRAN source code; an object file in the format produced by *fort77* **–c**; or a library of object files, in the format produced by archiving zero or more object files, using *ar*. Implementations may supply additional utilities that produce files in these formats. Additional input files are implementation-dependent.

A tab character encountered within the first six characters on a line of source code will cause the compiler to interpret the following character as if it were the seventh character on the line (that is, in column 7).

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *fort77*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TMPDIR
Determine the pathname that should override the default directory for temporary files, if any.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages. If more than one file operand ending in **.f** (or possibly other unspecified suffixes) is given, for each such file:

```
"%s:\n", <file>
```

may be written to allow identification of the diagnostic message with the appropriate input file.

This utility may produce warning messages about certain conditions that do not warrant returning an error (non-zero) exit value.

OUTPUT FILES

Object files, listing files and executable files are produced in unspecified formats.

EXTENDED DESCRIPTION

Standard Libraries

The *fort77* utility recognises the following **–l** operand for the standard library:

–l f This library contains all library functions referenced in the ANSI X3.9-1978 standard. This operand is not required to be present to cause a search of this library.

In the absence of options that inhibit invocation of the link editor, such as **–c,** the *fort77* utility will cause the equivalent of a **–l f** operand to be passed to the link editor as the last **–l** operand, causing it to be searched after all other object files and libraries are loaded.

It is unspecified whether the library **libf.a** exists as a regular file. The implementation may accept as **–l** operands names of objects that do not exist as regular files.

External Symbols

The FORTRAN compiler and link editor support the significance of external symbols up to a length of at least 31 bytes; case folding is permitted. The action taken upon encountering symbols exceeding the implementation-dependent maximum symbol length is unspecified.

The compiler and link editor support a minimum of 511 external symbols per source or object file, and a minimum of 4095 external symbols total. A diagnostic message is written to standard output if the implementation-dependent limit is exceeded; other actions are unspecified.

EXIT STATUS

The following exit values are returned:

0 Successful compilation or link edit.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When *fort77* encounters a compilation error, it will write a diagnostic to standard error and continue to compile other source code operands. It will return a non-zero exit status, but it is implementation-dependent whether an object module is created. If the link edit is unsuccessful, a diagnostic message will be written to standard error, and *fort77* will exit with a non-zero status.

APPLICATION USAGE

None.

EXAMPLES

The following are examples of usage:

`fort77 –o foo xyz.f`
Compiles **xyz.f** and creates the executable file **foo**.

`fort77 –c xyz.f`
Compiles **xyz.f** and creates the object file **xyz.o**.

`fort77 xyz.f`
Compiles **xyz.f** and creates the executable file **a.out**.

`fort77 xyz.f b.o`
Compiles **xyz.f**, links it with **b.o** and creates the executable **a.out**.

FUTURE DIRECTIONS

A compilation system based on FORTRAN-90 (ISO / IEC 1539: 1991) will be considered for a future issue; it may have a different utility name from *fort77*.

SEE ALSO

ar, *asa*, *c89*, *umask*.

CHANGE HISTORY

First released in Issue 4.

NAME

gencat – generate a formatted message catalogue

SYNOPSIS

EX `gencat` *`catfile msgfile...`*

DESCRIPTION

The *gencat* utility merges the message text source files *msgfile* into a formatted message catalogue *catfile*. The file *catfile* will be created if it does not already exist. If *catfile* does exist, its messages will be included in the new *catfile*. If set and message numbers collide, the new message text defined in *msgfile* will replace the old message text currently contained in *catfile*.

OPTIONS

None.

OPERANDS

The following operands are supported:

catfile A pathname of the formatted message catalogue. If – is specified, standard output is used. The format of the message catalogue produced is unspecified.

msgfile A pathname of a message text source file. If – is specified for an instance of *msgfile*, standard input is used. The format of message text source files is defined in the **EXTENDED DESCRIPTION**.

STDIN

The standard input is not used unless a *msgfile* operand is specified as –.

INPUT FILES

The input files are text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *gencat*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is not used unless the *catfile* operand is specified as –.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The format of a message text source file is defined as follows. Note that the fields of a message text source line are separated by a single blank character. Any other blank characters are considered as being part of the subsequent field.

$set *n comment*

This line specifies the set identifier of the following messages until the next **$set** or end-of-file appears. The *n* denotes the set identifier, which is defined as a number in the range [1, {NL_SETMAX}] (see **XSH** specification **<limits.h>**). Set identifiers must be presented in ascending order within a single source file but need not be contiguous. Any string following the set identifier is treated as a comment. If no **$set** directive is specified in a message text source file, all messages will be located in an implementation-defined default message set NL_SETD (see **<nl_types.h>** in the **XSH** specification).

$delset *n comment*

This line deletes message set *n* from an existing message catalogue. The *n* denotes the set number [1, {NL_SETMAX}]. Any string following the set number is treated as a comment.

$ *comment*

A line beginning with $ followed by a blank character is treated as a comment.

m message-text

The *m* denotes the message identifier, which is defined as a number in the range [1, {NL_MSGMAX}] (see **XSH** specification **<limits.h>**). The *message-text* is stored in the message catalogue with the set identifier specified by the last **$set** directive, and with message identifier *m*. If the *message-text* is empty, and a blank character field separator is present, an empty string is stored in the message catalogue. If a message source line has a message number, but neither a field separator nor *message-text*, the existing message with that number (if any) is deleted from the catalogue. Message identifiers must be in ascending order within a single set, but need not be contiguous. The length of *message-text* must be in the range [0, {NL_TEXTMAX}] (see **XSH** specification **<limits.h>**).

$quote *n*

This line specifies an optional quote character *c*, which can be used to surround *message-text* so that trailing spaces or null (empty) messages are visible in a message source line. By default, or if an empty **$quote** directive is supplied, no quoting of *message-text* will be recognised.

Empty lines in a message text source file are ignored. The effects of lines starting with any character other than those defined above are implementation-defined.

Text strings can contain the special characters and escape sequences defined in the following table:

Description	Symbol	Sequence
newline	NL(LF)	\n
horizontal tab	HT	\t
vertical-tab	VT	\v
backspace	BS	\b
carriage-return	CR	\r
form-feed	FF	\f
backslash	\	\\
bit pattern	*ddd*	*ddd*

The escape sequence *ddd* consists of backslash followed by one, two or three octal digits, which are taken to specify the value of the desired character. If the character following a backslash is not one of those specified, the backslash is ignored.

Backslash (\) followed by a newline character is also used to continue a string on the following line. Thus, the following two lines describe a single message string:

```
1 This line continues \
to the next line
```

which is equivalent to:

```
1 This line continues to the next line
```

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Message catalogues produced by *gencat* are binary encoded, meaning that their portability cannot be guaranteed between different types of machine. Thus, just as C programs need to be recompiled for each type of machine, so message catalogues must be recreated via *gencat*.

EXAMPLES

None.

FUTURE DIRECTIONS

The UniForum and POSIX committees are still working on a complete definition of message labels. The definition in this document will be aligned with the eventual outcome of their work.

SEE ALSO

iconv, the **XSH** specification description of **<limits.h>**.

CHANGE HISTORY

First released in Issue 3.

Issue 4

Format reorganised.

Internationalised environment variable support mandated.

NAME

get – get a version of an SCCS file (**DEVELOPMENT**)

SYNOPSIS

EX `get [-begkmlLpst][-c cutoff][-i list][-r SID][-x list] file...`

EX OB `get [-begkmpst][-c cutoff][-i list][ -l[p]][-r SID][-x list] file...`

DESCRIPTION

The *get* utility generates a text file from each named SCCS *file* according to the specifications given by its options.

The generated text is normally written into a file called the *g-file* whose name is derived from the SCCS filename by simply removing the leading **s.**.

OPTIONS

The *get* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except OB that, in the obsolescent form, **–l** option has an optional option-argument that cannot be presented as a separate argument (**–lp**). When the **–l** and **–p** options are both needed, the application must avoid ambiguity by giving them as separate arguments (**–l –p**), reversing their sequence (**–pl**) or separating them with other options in a single argument (such as **–ltb**). The following options are supported:

–r *SID* Indicate the SCCS Identification String (SID) of the version (delta) of an SCCS file to be retrieved. The table 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** option is also used), as a function of the SID specified.

–c *cutoff*

Indicate the *cutoff* date-time, in the form:

YY[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the generated text file. Units omitted from the date-time default to their maximum possible values; for example, **–c 7502** is equivalent to **–c 750228235959**.

Any number of non-numeric characters may separate the various 2-digit pieces of the *cutoff* date-time. This feature allows the user to specify a *cutoff* date in the form: **–c "77/2/2 9:22:25"**.

–e Indicate 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** option used in a *get* for a particular version (SID) of the SCCS file prevents further *get* commands from 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 a **–e** option is accidentally ruined in the process of editing, it may be regenerated by reexecuting the *get* command with the **–k** option in place of the **–e** option.

SCCS file protection specified via the ceiling, floor and authorised user list stored in the SCCS file is enforced when the **–e** option is used.

–b Use with the **–e** option to indicate that the new delta should have an SID in a new branch as shown in the table below. This option 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* Indicate 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 the table below. Partial SIDs are interpreted as shown in the ''SID Retrieved'' column of the table below.

–x *list* Indicate a *list* of deltas to be excluded (forced not to be applied) in the creation of the generated file. See the **–i** option for the *list* format.

–k Suppress replacement of identification keywords (see below) in the retrieved text by their value. The **–k** option is implied by the **–e** option.

–l Write a delta summary into an *l-file*.

–L Write a delta summary to standard output. All informative output that normally is written to standard output will be written to standard error instead, unless the **–s** option is used, in which case it is suppressed.

OB **–lp** Equivalent to **–L**.

–p Write the text retrieved from the SCCS file to the standard output. No *g-file* is created. All informative output that normally goes to the standard output goes to standard error instead, unless the **–s** option is used, in which case it disappears.

–s Suppress all informative output normally written to standard output. However, fatal error messages (which are always written to the standard error) remain unaffected.

–m Precede each text line retrieved from the SCCS file by the SID of the delta that inserted the text line in the SCCS file. The format is:

```
"%s\ts", <SID>, <text line>
```

–n Precede each generated text line with the **%M%** identification keyword value (see below). The format is:

```
"%s\ts", <%M% value>, <text line>
```

When both the **–m** and **–n** options are used, the *<text line>* is replaced by the **–m** option-generated format.

–g Suppress 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 Use to access the most recently created (top) delta in a given release (for example, **–r 1**), or release and level (for example, **–r 1.2**).

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *get* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only if the *file* operand is specified as –. Each line of the text file is interpreted as an SCCS pathname.

INPUT FILES

The SCCS files are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *get*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output (or standard error, if the –**p** option is used).

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

For each file processed, *get* writes to standard output the SID being accessed and the number of lines retrieved from the SCCS file, in the following format:

```
"%s\n%d\n", <SID>, <number of lines>
```

If the –**e** option is used, the SID of the delta to be made appears after the SID accessed and before the number of lines generated, in the POSIX locale:

```
"%s\nnew delta %s\n%d\n", <SID accessed>, <SID to be made>,
<number of lines>
```

If there is more than one named file or if a directory or standard input is named, each pathname is written before each of the lines shown in one of the preceding formats:

```
"\n%s:\n", <pathname>
```

OB If the –**L** (or –**lp**) option is used, a delta summary will be written following the format specified below for *l-files*.

If the –**i** option is used, included deltas are listed following the notation, in the POSIX locale:

```
"Included:\n"
```

If the **–x** option is used, excluded deltas are listed following the notation, in the POSIX locale:

```
"Excluded:\n"
```

OB If the **–p**, **–L** or **–lp** options are specified, the standard output consists of the text retrieved from the SCCS file.

STDERR

The standard error is used only for diagnostic messages, except if the **–p**, **–L** or OB **–lp** options are specified, it includes all informative messages normally sent to standard output.

OUTPUT 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 filename is formed from the SCCS file name: the last component of all SCCS filenames 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, for **s.xyz.c**, the auxiliary filenames 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** option 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** option is used or implied, it is writable 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** option 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:

```
"%c%c%c␣%s\t%s␣%s\n", <code1>, <code2>, <code3>, <SID>, <date-time>,
<login>
```

where the entries are:

<*code1*> A space character if the delta was applied; * otherwise.

<*code2*> A space character if the delta was applied or was not applied and ignored; * if the delta was not applied and was not ignored.

<*code3*> A character indicating a special reason why the delta was or was not applied:

I Included.

X Excluded.

C Cut off (by a **–c** option).

<*date-time*>
Date and time (using the *date* utility's %y/%m/%d %T format) of creation.

<*login*> Login name of person who created *delta*.

The comments and MR data follow on subsequent lines, indented one tab character. A blank line terminates each entry.

The *p-file* is used to pass information resulting from a *get* with a **–e** option along to *delta*. Its contents are also used to prevent a subsequent execution of *get* with a **–e** option 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 writable by owner only, and it is owned by the effective user. Each line in the *p-file* has the following format:

```
"%s␣%s␣%s␣%s␣%s%s\n", <g-file SID>, <SID of new delta>,
<login-name of real user>, <date-time>, <i-value>, <x-value>
```

where <*i-value*> is the value of the *list* option-argument to **–i** (or null) and <*x-value*> is the value of the *list* option-argument to **–x** (or null). 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 (that is, *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.

EXTENDED DESCRIPTION

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 successor in release > R and R exists	R.mL	R.mL.(mB+1).1
R.L	no	No trunk successor	R.L	R.(L+1)
R.L	yes	No trunk successor	R.L	R.L.(mB+1).1
R.L	–	Trunk successor in release ≥ R	R.L	R.L.(mB+1).1
R.L.B	no	No branch successor	R.L.B.mS	R.L.B.(mS+1)
R.L.B	yes	No branch successor	R.L.B.mS	R.L.(mB+1).1
R.L.B.S	no	No branch successor	R.L.B.S	R.L.B.(S+1)
R.L.B.S	yes	No branch successor	R.L.B.S	R.L.(mB+1).1
R.L.B.S	–	Branch successor	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 (that is, 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, non-existent, release R.

*** This is used to force creation of the first delta in a new release.

† The **–b** option 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:

%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*% or %*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 filename.
%P% SCCS absolute pathname.
%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 four-character string @(#) recognisable by *what*.
%W% A shorthand notation for constructing *what* strings:

```
%W% = %Z%%M%<tab>%I%
```

%A% Another shorthand notation for constructing *what* strings:

```
%A% = %Z%%Y% %M% %I%%Z%
```

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

The **–lp** option may be withdrawn from a future issue.

SEE ALSO

admin, *delta*, *prs*, *what*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

NAME

getconf – get configuration values

SYNOPSIS

`getconf` *`system_var`*

`getconf` *`path_var pathname`*

DESCRIPTION

In the first synopsis form, the *getconf* utility will write to the standard output the value of the variable specified by the *system_var* operand.

In the second synopsis form, the *getconf* utility will write to the standard output the value of the variable specified by the *path_var* operand for the path specified by the *pathname* operand.

The value of each configuration variable will be determined as if it were obtained by calling the function from which it is defined to be available by this standard or by the **XSH** specification (see **OPERANDS**). The value will reflect conditions in the current operating environment.

OPTIONS

None.

OPERANDS

The following operands are supported:

path_var

A name of a configuration variable whose value is available from the **XSH** specification *pathconf*( ) function. All of the values in the following table are supported and the implementation may add other local values:

LINK_MAX	NAME_MAX	_POSIX_CHOWN_RESTRICTED
MAX_CANON	PATH_MAX	_POSIX_NO_TRUNC
MAX_INPUT	PIPE_BUF	_POSIX_VDISABLE

pathname

A pathname for which the variable specified by *path_var* is to be determined.

system_var

A name of a configuration variable whose value is available from the **XSH** specification *confstr*( ) or *sysconf*( ) functions. All of the values in the following table are supported and the implementation may add other local values:

ARG_MAX	NL_TEXTMAX	POSIX2_C_DEV
BC_BASE_MAX	OPEN_MAX	POSIX2_C_VERSION
BC_DIM_MAX	_POSIX_ARG_MAX	POSIX2_EXPR_NEST_MAX
BC_SCALE_MAX	_POSIX_CHILD_MAX	POSIX2_FORT_DEV
BC_STRING_MAX	_POSIX_JOB_CONTROL	POSIX2_FORT_RUN
CHAR_BIT	_POSIX_LINK_MAX	POSIX2_LINE_MAX
CHAR_MAX	_POSIX_MAX_CANON	POSIX2_LOCALEDEF
CHAR_MIN	_POSIX_MAX_INPUT	POSIX2_RE_DUP_MAX
CHILD_MAX	_POSIX_NAME_MAX	POSIX2_SW_DEV
CLK_TCK	_POSIX_NGROUPS_MAX	POSIX2_UPE
COLL_WEIGHTS_MAX	_POSIX_OPEN_MAX	POSIX2_VERSION

	CS_PATH	_POSIX_PATH_MAX	RE_DUP_MAX
	EXPR_NEST_MAX	_POSIX_PIPE_BUF	SCHAR_MAX
	INT_MAX	_POSIX_SAVED_IDS	SCHAR_MIN
	INT_MIN	_POSIX_SSIZE_MAX	SHRT_MAX
	LINE_MAX	_POSIX_STREAM_MAX	SHRT_MIN
	LONG_MAX	_POSIX_TZNAME_MAX	SSIZE_MAX
	LONG_MIN	_POSIX_VERSION	STREAM_MAX
	MB_LEN_MAX	POSIX2_BC_BASE_MAX	TZNAME_MAX
	NGROUPS_MAX	POSIX2_BC_DIM_MAX	UCHAR_MAX
	NL_ARGMAX	POSIX2_BC_SCALE_MAX	UINT_MAX
	NL_LANGMAX	POSIX2_BC_STRING_MAX	ULONG_MAX
	NL_MAX	POSIX2_CHAR_TERM	USHRT_MAX
	NL_MSGMAX	POSIX2_COLL_WEIGHTS_MAX	
	NL_SETMAX	POSIX2_C_BIND	
EX	CHARCLASS_NAME_MAX	NL_TEXTMAX	_XOPEN_VERSION
	LONG_BIT	NZERO	_XOPEN_XCU_VERSION
	NL_ARGMAX	TMP_MAX	_XOPEN_XPG2
	NL_LANGMAX	WORD_BIT	_XOPEN_XPG3
	NL_MAX	_XOPEN_CRYPT	_XOPEN_XPG4
	NL_MSGMAX	_XOPEN_ENH_I18N	
	NL_SETMAX	_XOPEN_SHM	
UX	ATEXIT_MAX	PAGESIZE	_XOPEN_UNIX
	IOV_MAX	PAGE_SIZE	

The symbol PATH also is recognised, yielding the same value as the *confstr*( ) name value CS_PATH.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *getconf*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If the specified variable is defined on the system and its value is described to be available from the **XSH** specification *confstr*( ) function, its value will be written in the following format:

```
"%s\n", <value>
```

Otherwise, if the specified variable is defined on the system, its value will be written in the following format:

```
"%d\n", <value>
```

If the specified variable is valid, but is undefined on the system, *getconf* will write using the following format:

```
"undefined\n"
```

If the variable name is invalid or an error occurs, nothing will be written to standard output.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The specified variable is valid and information about its current state was written successfully.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

This example illustrates the value of {NGROUPS_MAX}:

```
getconf NGROUPS_MAX
```

This example illustrates the value of {NAME_MAX} for a specific directory:

```
getconf NAME_MAX /usr
```

This example shows how to deal more carefully with results that might be unspecified:

```
if value=$(getconf PATH_MAX /usr); then
       if [ "$value" = "undefined" ]; then
              echo PATH_MAX in /usr is infinite.
       else
              echo PATH_MAX in /usr is $value.
       fi
else
       echo Error in getconf.
fi
```

Note that:

```
sysconf(_SC_POSIX_C_BIND);
```

and:

```
system("getconf POSIX2_C_BIND");
```

in a C program could give different answers. The *sysconf*( ) call supplies a value that corresponds to the conditions when the program was either compiled or executed, depending on the implementation; the *system*( ) call to *getconf* always supplies a value corresponding to conditions when the program is executed.

FUTURE DIRECTIONS

None.

SEE ALSO

The **XSH** specification description of *confstr*( ), *pathconf*( ), *sysconf*( ).

CHANGE HISTORY

First released in Issue 4.

Issue 4, Version 2

The following changes are made in the table of values for *system_var*:

- Names beginning with POSIX_ are changed to begin with _POSIX_.
- Names beginning with XOPEN_ are changed to begin with _XOPEN_.
- MN_NMAX is changed to NL_MAX.
- NL_SET_MAX is changed to NL_SETMAX.
- NL_TEXT_MAX is changed to NL_TEXTMAX.
- The _XOPEN_CRYPT, _XOPEN_ENH_I18N and _XOPEN_SHM configuration variables are added to the list.

NAME

getopts – parse utility options

SYNOPSIS

`getopts` *`optstring name`* **[***`arg...`***]**

DESCRIPTION

The *getopts* utility can be used to retrieve options and option-arguments from a list of parameters. It supports the utility argument syntax guidelines 3 to 10, inclusive, described in the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

Each time it is invoked, the *getopts* utility places the value of the next option in the shell variable specified by the *name* operand and the index of the next argument to be processed in the shell variable *OPTIND*. Whenever the shell is invoked, *OPTIND* will be initialised to 1.

When the option requires an option-argument, the *getopts* utility will place it in the shell variable *OPTARG*. If no option was found, or if the option that was found does not have an option-argument, *OPTARG* will be unset.

If an option character not contained in the *optstring* operand is found where an option character is expected, the shell variable specified by *name* will be set to the question-mark (?) character. In this case, if the first character in *optstring* is a colon (:), the shell variable *OPTARG* will be set to the option character found, but no output will be written to standard error; otherwise, the shell variable *OPTARG* will be unset and a diagnostic message will be written to standard error. This condition is considered to be an error detected in the way arguments were presented to the invoking application, but is not an error in *getopts* processing.

If an option-argument is missing:

- If the first character of *optstring* is a colon, the shell variable specified by *name* will be set to the colon character and the shell variable *OPTARG* will be set to the option character found.
- Otherwise, the shell variable specified by *name* will be set to the question-mark character, the shell variable *OPTARG* will be unset, and a diagnostic message will be written to standard error. This condition is considered to be an error detected in the way arguments were presented to the invoking application, but is not an error in *getopts* processing; a diagnostic message will be written as stated, but the exit status will be zero.

When the end of options is encountered, the *getopts* utility will exit with a return value greater than zero; the shell variable *OPTIND* will be set to the index of the first non-option-argument, where the first `--` argument is considered to be an option-argument if there are no other non-option-arguments appearing before it, or the value $# + 1 if there are no non-option-arguments; the *name* variable will be set to the question-mark character. Any of the following identifies the end of options: the special option `--`, finding an argument that does not begin with a `-`, or encountering an error.

The shell variables *OPTIND* and *OPTARG* are local to the caller of *getopts* and are not exported by default.

The shell variable specified by the *name* operand, *OPTIND* and *OPTARG* affect the current shell execution environment; see Section 2.12 on page 63.

If the application sets *OPTIND* to the value 1, a new set of parameters can be used: either the current positional parameters or new *arg* values. Any other attempt to invoke *getopts* multiple times in a single shell execution environment with parameters (positional parameters or *arg* operands) that are not the same in all invocations, or with an *OPTIND* value modified to be a value other than 1, produces unspecified results.

OPTIONS

None.

OPERANDS

The following operands are supported:

optstring
: A string containing the option characters recognised by the utility invoking *getopts*. If a character is followed by a colon, the option will be expected to have an argument, which should be supplied as a separate argument. Applications should specify an option character and its option-argument as separate arguments, but *getopts* will interpret the characters following an option character requiring arguments as an argument whether or not this is done. An explicit null option-argument need not be recognised if it is not supplied as a separate argument when *getopts* is invoked. (See also the **XSH** specification *getopt*( ) function.) The characters question-mark and colon must not be used as option characters by an application. The use of other option characters that are not alphanumeric produces unspecified results. If the option-argument is not supplied as a separate argument from the option character, the value in *OPTARG* will be stripped of the option character and the –. The first character in *optstring* will determine how *getopts* will behave if an option character is not known or an option-argument is missing.

name
: The name of a shell variable that will be set by the *getopts* utility to the option character that was found.

The *getopts* utility by default will parse positional parameters passed to the invoking shell procedure. If *arg*s are given, they will be parsed instead of the positional parameters.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *getopts*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

OPTIND

This variable will be used by the *getopts* utility as the index of the next argument to be processed.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Whenever an error is detected and the first character in the *optstring* operand is not a colon (:), a diagnostic message will be written to standard error with the following information in an unspecified format:

- The invoking program name will be identified in the message. The invoking program name will be the value of the shell special parameter 0 (see Section 2.5.2 on page 27) at the time the *getopts* utility is invoked. A name equivalent to:

  ```
  basename "$0"
  ```

  may be used.

- If an option is found that was not specified in *optstring*, this error will be identified and the invalid option character will be identified in the message.

- If an option requiring an option-argument is found, but an option-argument is not found, this error will be identified and the invalid option character will be identified in the message.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 An option, specified or unspecified by *optstring*, was found.
>0 The end of options was encountered or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *getopts* affects the current shell execution environment, it is generally provided as a shell regular built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(getopts abc value "$@")
nohup getopts ...
find . -exec getopts ... \;
```

it will not affect the shell variables in the caller's environment.

Note that shell functions share *OPTIND* with the calling shell even though the positional parameters are changed. Functions that want to use *getopts* to parse their arguments will usually want to save the value of *OPTIND* on entry and restore it before returning. However, there will be cases when a function will want to change *OPTIND* for the calling shell.

EXAMPLES

The following example script parses and displays its arguments:

```
aflag=
bflag=
while getopts ab: name
do
       case $name in
       a)     aflag=1;;
       b)     bflag=1
              bval="$OPTARG";;
       ?)     printf "Usage: %s: [-a] [-b value] args\n" $0
              exit 2;;
       esac
done
if [ ! -z "$aflag" ]; then
       printf "Option -a specified\n"
fi
if [ ! -z "$bflag" ]; then
       printf 'Option -b "%s" specified\n' "$bval"
fi
shift $(($OPTIND - 1))
printf "Remaining arguments are: %s\n" "$*"
```

FUTURE DIRECTIONS

None.

SEE ALSO

The **XSH** specification description of *getopt*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

grep – search a file for a pattern

SYNOPSIS

```
grep [ -E| -F][ -c| -l| -q ][-insvx] -e pattern_list
[-f pattern_file]...[file...]

grep [ -E| -F][ -c| -l| -q ][-insvx][-e pattern_list
-f pattern_file...[file...]

grep [ -E| -F][ -c| -l| -q ][-insvx] pattern_list[file...]
```

DESCRIPTION

The *grep* utility searches the input files, selecting lines matching one or more patterns; the types of patterns are controlled by the options specified. The patterns are specified by the **–e** option, **–f** option, or the *pattern_list* operand. The *pattern_list*'s value consists of one or more patterns separated by newline characters; the *pattern_file*'s contents consist of one or more patterns terminated by newline characters. By default, an input line will be selected if any pattern, treated as an entire basic regular expression (BRE) as described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**, matches any part of the line; a null BRE will match every line. By default, each selected input line will be written to the standard output.

Regular expression matching will be based on text lines. Since a newline character separates or terminates patterns (see the **–e** and **–f** options below), regular expressions cannot contain a newline character. Similarly, since patterns are matched against individual lines of the input, there is no way for a pattern to match a newline character found in the input.

OPTIONS

The *grep* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–E Match using extended regular expressions. Treat each pattern specified as an ERE, as described in the **XBD** specification, **Section 7.4**, **Extended Regular Expressions**. If any entire ERE pattern matches an input line, the line will be matched. A null ERE matches every line.

–F Match using fixed strings. Treat each pattern specified as a string instead of a regular expression. If an input line contains any of the patterns as a contiguous sequence of bytes, the line will be matched. A null string matches every line.

–c Write only a count of selected lines to standard output.

–e *pattern_list*
Specify one or more patterns to be used during the search for input. Patterns in *pattern_list* must be separated by a newline character. A null pattern can be specified by two adjacent newline characters in *pattern_list*. Unless the **–E** or **–F** option is also specified, each pattern will be treated as a BRE, as described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**. Multiple **–e** and **–f** options are accepted by the *grep* utility. All of the specified patterns are used when matching lines, but the order of evaluation is unspecified.

–f *pattern_file*
Read one or more patterns from the file named by the pathname *pattern_file*. Patterns in *pattern_file* are terminated by a newline character. A null pattern can be specified by an empty line in *pattern_file*. Unless the **–E** or **–F** option is also specified, each pattern will be treated as a BRE, as described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**.

–i Perform pattern matching in searches without regard to case. See the **XBD** specification, **Section 7.2**, **Regular Expression General Requirements**.

–l (The letter ell.) Write only the names of files containing selected lines to standard output. Pathnames are written once per file searched. If the standard input is searched, a pathname of "**(standard input)**" will be written, in the POSIX locale. In other locales, **standard input** may be replaced by something more appropriate in those locales.

–n Precede each output line by its relative line number in the file, each file starting at line 1. The line number counter will be reset for each file processed.

–q Quiet. Do not write anything to the standard output, regardless of matching lines. Exit with zero status if an input line is selected.

–s Suppress the error messages ordinarily written for non-existent or unreadable files. Other error messages will not be suppressed.

–v Select lines not matching any of the specified patterns. If the **–v** option is not specified, selected lines will be those that match any of the specified patterns.

–x Consider only input lines that use all characters in the line to match an entire fixed string or regular expression to be matching lines.

OPERANDS

The following operands are supported:

pattern Specify one or more patterns to be used during the search for input. This operand is treated as if it were specified as **–e** *pattern_list*.

file A pathname of a file to be searched for the patterns. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *grep*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within regular expressions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If the **–l** option is in effect, and the **–q** option is not, the following will be written for each file containing at least one selected input line:

```
"%s\n", file
```

Otherwise, if more than one *file* argument appears, and **–q** is not specified, the *grep* utility will prefix each output line by:

```
"%s:", file
```

The remainder of each output line depends on the other options specified:

- If the **–c** option is in effect, the remainder of each output line will contain:

  ```
  "%d\n", <count>
  ```

- Otherwise, if **–c** is not in effect and the **–n** option is in effect, the following will be written to standard output:

  ```
  "%d:", <line number>
  ```

- Finally, the following will be written to standard output:

  ```
  "%s", <selected-line contents>
  ```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 One or more lines were selected.
1 No lines were selected.
>1 An error occurred.

CONSEQUENCES OF ERRORS
If the **–q** option is specified, the exit status will be zero if an input line is selected, even if an error was detected. Otherwise, default actions will be performed.

APPLICATION USAGE
Care should be taken when using characters in *pattern_list* that may also be meaningful to the command interpreter. It is safest to enclose the entire *pattern_list* argument in single quotes:

```
'...'
```

The **–e** *pattern_list* option has the same effect as the *pattern_list* operand, but is useful when *pattern_list* begins with the hyphen delimiter. It is also useful when it is more convenient to provide multiple patterns as separate arguments.

Multiple **–e** and **–f** options are accepted and *grep* will use all of the patterns it is given while matching input text lines. (Note that the order of evaluation is not specified. If an implementation finds a null string as a pattern, it is allowed to use that pattern first, matching every line, and effectively ignore any other patterns.)

The **–q** option provides a means of easily determining whether or not a pattern (or string) exists in a group of files. When searching several files, it provides a performance improvement (because it can quit as soon as it finds the first match) and requires less care by the user in choosing the set of files to supply as arguments (because it will exit zero if it finds a match even if *grep* detected an access or read error on earlier file operands).

EXAMPLES

1. To find all uses of the word Posix (in any case) in file **text.mm** and write with line numbers:

```
grep -i -n posix text.mm
```

2. To find all empty lines in the standard input:

```
grep ^$
```
or:
```
grep -v .
```

3. Both of the following commands print all lines containing strings **abc** or **def** or both:

```
grep -E 'abc
def'
grep -F 'abc
def'
```

4. Both of the following commands print all lines matching exactly **abc** or **def**:

```
grep -E '^abc$
^def$'
grep -F -x 'abc
def'
```

FUTURE DIRECTIONS

None.

SEE ALSO

egrep, *fgrep*, *sed*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

hash – remember or report utility locations

SYNOPSIS

EX `hash [`*`utility`*`...]`

EX `hash -r`

DESCRIPTION

The *hash* utility affects the way the current shell environment remembers the locations of utilities found as described in **Command Search and Execution** on page 47. Depending on the arguments specified, it adds utility locations to its list of remembered locations or it purges the contents of the list. When no arguments are specified, it reports on the contents of the list.

Utilities provided as built-ins to the shell are not reported by *hash*.

OPTIONS

The *hash* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–r Forget all previously remembered utility locations.

OPERANDS

The following operand is supported:

utility The name of a utility to be searched for and added to the list of remembered locations. If *utility* contains one or more slashes, the results are unspecified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *hash*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the location of *utility*, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output of *hash* is used when no arguments are specified. Its format is unspecified, but includes the pathname of each utility in the list of remembered locations for the current shell environment. This list consists of those utilities named in previous *hash* invocations that have been invoked, and may contain those invoked and found through the normal command search process.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *hash* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in a separate utility execution environment, such as one of the following:

```
nohup hash -r
find . -type f | xargs hash
```

it will not affect the command search process of the caller's environment.

The *hash* utility may be implemented as an alias, for example, *alias* –**t** –, in which case utilities found through normal command search will not be listed by the *hash* command.

The effects of *hash* –**r** can also be achieved portably by resetting the value of *PATH*; in the simplest form, this can be:

```
PATH="$PATH"
```

The use of *hash* with *utility* names is unnecessary for most applications, but may provide a performance improvement on a few implementations; normally, the hashing process is included by default.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

Command Search and Execution on page 47.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Relocated from the *sh* description to reflect its status as a regular built-in utility.

NAME

head – copy the first part of files

SYNOPSIS

```
head [-n number][file...]
```

OB

```
head [number][file...]
```

DESCRIPTION

The *head* utility will copy its input files to the standard output, ending the output for each file at a designated point.

OB Copying will end at the point in each input file indicated by the **–n** *number* option (or the *–number* argument). The option-argument *number* will be counted in units of lines.

OPTIONS

OB The *head* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the obsolescent version accepts multi-character numeric options.

The following options are supported:

–n *number*
: The first *number* lines of each input file will be copied to standard output. The *number* option-argument must be a positive decimal integer.

OB *–number*
: The *number* argument is a positive decimal integer with the same effect as the **–n** *number* option.

If no options are specified, *head* will act as if **–n 10** had been specified.

OPERANDS

The following operand is supported:

file A pathname of an input file. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

Input files must be text files, but the line length is not restricted to {LINE_MAX} bytes.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *head*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output will contain designated portions of the input files.

If multiple *file* operands are specified, *head* will precede the output for each with the header:

```
"\n==> %s <==\n", <pathname>
```

except that the first header written will not include the initial newline character.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
To write the first ten lines of all files (except those with a leading period) in the directory:

```
head *
```

FUTURE DIRECTIONS
The obsolescent *–number* form may be withdrawn from a future issue. Applications should use the **–n** *number* option.

SEE ALSO
sed, *tail*.

CHANGE HISTORY
First released in Issue 4.

NAME

iconv – codeset conversion

SYNOPSIS

EX `iconv -f fromcode -t tocode [file...]`

DESCRIPTION

The *iconv* utility converts the encoding of characters in *file* from one codeset to another and writes the results to standard output.

Character encodings in either codeset may include single-byte values (for example, for the ISO 8859-1: 1987 standard characters) or multi-byte values (for example, for certain characters in the ISO 6937: 1983 standard). The results of specifying invalid characters in the input stream (either those that are not valid members of the *fromcode* or those that have no corresponding value in *tocode*) are specified in the system documentation.

OPTIONS

The *iconv* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–**f** *fromcode*
: Identify the codeset of the input file. Valid values for *fromcode* are specified in the system documentation.

–**t** *tocode*
: Identify the codeset to be used for the output file. Valid values for *tocode* are specified in the system documentation.

OPERANDS

The following operands are supported:

file A pathname of the input file to be translated. If *file* is omitted, the standard input is used.

STDIN

The standard input is used only if the *file* operand is omitted.

INPUT FILES

The input file is a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *iconv*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments). During translation of the file, this variable is superseded by the use of the *fromcode* option-argument.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output is a text file containing the translated data.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
The following example converts the contents of file **mail.x400** from the ISO 6937: 1983 standard codeset to the ISO 8859-1: 1987 standard codeset, and stores the results in file **mail.local**:

```
iconv –f IS6937 –t IS8859 mail.x400 > mail.local
```

FUTURE DIRECTIONS
None.

SEE ALSO
gencat.

CHANGE HISTORY
First released in Issue 3.

Issue 4
Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

id – return user identity

SYNOPSIS

```
id [user]
id -G[-n] [user]
id -g[-nr] [user]
id -u[-nr] [user]
```

DESCRIPTION

If no *user* operand is provided, the *id* utility will write the user and group IDs and the corresponding user and group names of the invoking process to standard output. If the effective and real IDs do not match, both will be written. If multiple groups are supported by the underlying system (see the description of {NGROUPS_MAX} in the **XSH** specification), the supplementary group affiliations of the invoking process also will be written.

If a *user* operand is provided and the process has the appropriate privileges, the user and group IDs of the selected user will be written. In this case, effective IDs will be assumed to be identical to real IDs. If the selected user has more than one allowable group membership listed in the group database, these will be written in the same manner as the supplementary groups described in the preceding paragraph.

OPTIONS

The *id* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–G Output all different group IDs (effective, real and supplementary) only, using the format "%u\n". If there is more than one distinct group affiliation, output each such affiliation, using the format " %u", before the newline character is output.

–g Output only the effective group ID, using the format "%u\n".

–n Output the name in the format "%s" instead of the numeric ID using the format "%u".

–r Output the real ID instead of the effective ID.

–u Output only the effective user ID, using the format "%u\n".

OPERANDS

The following operand is supported:

user The login name for which information is to be written.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *id*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The following formats will be used when the LC_MESSAGES locale category specifies the POSIX locale. In other locales, the strings **uid, gid, euid, egid** and **groups** may be replaced with more appropriate strings corresponding to the locale.

```
"uid=%u(%s) gid=%u(%s)\n", <real user ID>, <user-name>,
<real group ID>, <group-name>
```

If the effective and real user IDs do not match, the following will be inserted immediately before the `\n` character in the previous format:

```
" euid=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<effective user ID>, <effective user-name>
```

If the effective and real group IDs do not match, the following will be inserted directly before the `\n` character in the format string (and after any addition resulting from the effective and real user IDs not matching):

```
" egid=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<effective group-ID>, <effective group name>
```

If the process has supplementary group affiliations or the selected user is allowed to belong to multiple groups, the first will be added directly before the newline character in the format string:

```
" groups=%u(%s)"
```

with the following arguments added at the end of the argument list:

```
<supplementary group ID>, <supplementary group name>
```

and the necessary number of the following added after that for any remaining supplementary group IDs:

```
",%u(%s)"
```

and the necessary number of the following arguments added at the end of the argument list:

<supplementary group ID>, *<supplementary group name>*

If any of the user ID, group ID, effective user ID, effective group ID or supplementary/multiple group IDs cannot be mapped by the system into printable user or group names, the corresponding (**%s**) and name argument will be omitted from the corresponding format string.

When any of the options are specified, the output format will be as described under **OPTIONS.**

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Output produced by the **−G** option and by the default case could potentially produce very long lines on systems that support large numbers of supplementary groups. (On systems with user and group IDs that are 32-bit integers and with group names with a maximum of 8 bytes per name, 93 supplementary groups plus distinct effective and real group and user IDs could theoretically overflow the 2048-byte {LINE_MAX} text file line limit on the default output case. It would take about 186 supplementary groups to overflow the 2048-byte barrier using *id* **−G**.) This is not expected to be a problem in practice, but in cases where it is a concern, applications should consider using *fold* **−s** before postprocessing the output of *id*.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

fold, *logname*, *who*, the **XSH** specification description of *getgid*( ), *getgroups*( ), *getuid*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO/IEC 9945-2: 1993 standard.

NAME

jobs – display status of jobs in the current session

SYNOPSIS

JC `jobs [ -l| -p][job_id...]`

DESCRIPTION

The *jobs* utility displays the status of jobs that were started in the current shell environment; see Section 2.12 on page 63.

When *jobs* reports the termination status of a job, the shell removes its process ID from the list of those ‘‘known in the current shell execution environment’’; see Section 2.9.3 on page 50.

OPTIONS

The *jobs* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–l (The letter ell.) Provide more information about each job listed. This information includes the job number, current job, process group ID, state and the command that formed the job.

–p Display only the process IDs for the process group leaders of the selected jobs.

By default, the *jobs* utility displays the status of all stopped jobs, running background jobs and all jobs whose status has changed and have not been reported by the shell.

OPERANDS

The following operand is supported:

job_id Specifies the jobs for which the status is to be displayed. If no *job_id* is given, the status information for all jobs will be displayed. The format of *job_id* is described in the entry for **job control job ID** in the **XBD** specification, **Chapter 2**, **Glossary**.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *jobs*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If the –**p** option is specified, the output consists of one line for each process ID:

```
"%d\n", <process ID>
```

Otherwise, if the –**l** option is not specified, the output is a series of lines of the form:

```
"[%d] %c %s %s\n", <job-number>, <current>, <state>, <command>
```

where the fields are as follows:

<*current*>
The character + identifies the job that would be used as a default for the *fg* or *bg* utilities; this job can also be specified using the *job_id* **%+** or **%%**. The character – identifies the job that would become the default if the current default job were to exit; this job can also be specified using the *job_id* **%**–. For other jobs, this field is a space character. At most one job can be identified with + and at most one job can be identified with –. If there is any suspended job, then the current job will be a suspended job. If there are at least two suspended jobs, then the previous job will also be a suspended job.

<*job-number*>
A number that can be used to identify the process group to the *wait*, *fg*, *bg* and *kill* utilities. Using these utilities, the job can be identified by prefixing the job number with %.

<*state*> One of the following strings (in the POSIX Locale):

Running
Indicates that the job has not been suspended by a signal and has not exited.

Done Indicates that the job completed and returned exit status zero.

Done(*code***)**
Indicates that the job completed normally and that it exited with the specified non-zero exit status, *code*, expressed as a decimal number.

Stopped
Stopped (SIGTSTP)
Indicates that the job was suspended by the SIGTSTP signal.

Stopped (SIGSTOP)
Indicates that the job was suspended by the SIGSTOP signal.

Stopped (SIGTTIN)
Indicates that the job was suspended by the SIGTTIN signal.

Stopped (SIGTTOU)
Indicates that the job was suspended by the SIGTTOU signal.

The implementation may substitute the string **Suspended** in place of **Stopped**. If the job was terminated by a signal, the format of <*state*> is unspecified, but it will be visibly distinct from all of the other <*state*> formats shown here and will indicate the name or description of the signal causing the termination.

<*command*>
The associated command that was given to the shell.

If the **–l** option is specified, a field containing the process group ID is inserted before the <*state*> field. Also, more processes in a process group may be output on separate lines, using only the process ID and <*command*> fields.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The **–p** option is the only way to find out the process group of a job because different implementations have different strategies for defining the process group of the job. Usage such as $(jobs –p) provides a way of referring to the process group of the job in an implementation-independent way.

The *jobs* utility will not work as expected when it is operating in its own utility execution environment because that environment will have no applicable jobs to manipulate. See the **APPLICATION USAGE** section for *bg*. For this reason, *jobs* is generally implemented as a shell regular built-in.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
bg, *fg*, *kill*, *wait*.

CHANGE HISTORY
First released in Issue 4.

NAME

join – relational database operator

SYNOPSIS

```
join [ -a file_number | -v file_number ][-e string][-o list][-t char]
[-1 field][-2 field] file1 file2
```

OB

```
join [-a file_number][-e string][-j field][-j1 field][-j2 field]
[-o list...][-t char][-t char] file1 file2
```

DESCRIPTION

The *join* utility will perform an equality join on the files *file1* and *file2*. The joined files will be written to the standard output.

The join field is a field in each file on which the files are compared. By default, *join* writes one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line by default will consist of the join field, then the remaining fields from *file1*, then the remaining fields from *file2*. This format can be changed by using the –**o** option (see below). The –**a** option can be used to add unmatched lines to the output. The –**v** option can be used to output only unmatched lines.

By default, the files *file1* and *file2* should be ordered in the collating sequence of *sort* –**b** on the fields on which they are to be joined, by default the first in each line. All selected output will be written in the same collating sequence.

The default input field separators will be blank characters. In this case, multiple separators will count as one field separator, and leading separators will be ignored. The default output field separator will be a space character.

The field separator and collating sequence can be changed by using the –**t** option (see below).

If the input files are not in the appropriate collating sequence, the results are unspecified.

OPTIONS

OB The *join* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The obsolescent version does not follow the utility argument syntax guidelines: the –**j1** and –**j2** options are multi-character options and the –**o** option takes multiple arguments.

The following options are supported:

–**a** *file_number*
: Produce a line for each unpairable line in file *file_number*, where *file_number* is 1 or 2, in addition to the default output. If both –**a 1** and –**a 2** are specified, all unpairable lines will be output.

–**e** *string*
: Replace empty output fields in the list selected by –**o** with the string *string*.

OB –**j** *field* Equivalent to: –**1** *field* –**2** *field*.

OB –**j1** *field* Equivalent to: –**1** *field*.

OB –**j2** *field* Equivalent to: –**2** *field*.

–**o** *list* Construct the output line to comprise the fields specified in *list*, each element of which has one of the following two forms:

- *file_number.field*, where *file_number* is a file number and *field* is a decimal integer field number

- 0 (zero), representing the join field.

The elements of *list* are either comma- or blank-separated, as specified in Guideline 8 of the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The fields specified by *list* will be written for all selected output lines. Fields selected by *list* that do not appear in the input will be treated as empty output fields. (See the **–e** option.) Only specifically requested fields are written. The *list* must be a single command line
OB argument. However, as an obsolescent feature, the argument *list* can be multiple arguments on the command line.

–t *char* Use character *char* as a separator, for both input and output. Every appearance of *char* in a line will be significant. When this option is specified, the collating sequence should be the same as *sort* without the **–b** option.

–v *file_number*
Instead of the default output, produce a line only for each unpairable line in *file_number*, where *file_number* is 1 or 2. If both **–v 1** and **–v 2** are specified, all unpairable lines will be output.

–1 *field* Join on the *field*th field of file 1. Fields are decimal integers starting with 1.

–2 *field* Join on the *field*th field of file 2. Fields are decimal integers starting with 1.

OPERANDS

The following operands are supported:

file1
file2 A pathname of a file to be joined. If either of the *file1* or *file2* operands is –, the standard input is used in its place.

STDIN

The standard input will be used only if the *file1* or *file2* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *join*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale of the collating sequence *join* expects to have been used when the input files were sorted.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *join* utility output will be a concatenation of selected character fields. When the **–o** option is not specified, the output will be:

```
"%s%s%s\n", <join field>, <other file1 fields>,
<other file2 fields>
```

If the join field is not the first field in a file, the *<other file fields>* for that file are:

```
<fields preceding join field>, <fields following join field>
```

When the **–o** option is specified, the output format will be:

```
"%s\n", <concatenation of fields>
```

where the concatenation of fields is described by the **–o** option, above.

For either format, each field (except the last) will be written with its trailing separator character. If the separator is the default (blank characters), a single space character will be written after each field (except the last).

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All input files were output successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Pathnames consisting of numeric digits or of the form *string.string* should not be specified directly following the **–o** list.

EXAMPLES
The **–o 0** field essentially selects the union of the join fields. For example, given file **phone**:

```
!Name           Phone Number
Don             +1 123-456-7890
Hal             +1 234-567-8901
Yasushi         +2 345-678-9012
```

and file **fax**:

```
!Name           Fax Number
Don             +1 123-456-7899
Keith           +1 456-789-0122
Yasushi         +2 345-678-9011
```

(where the large expanses of white space are meant to each represent a single tab character), the command:

```
join -t "<tab>" -a 1 -a 2 -e '(unknown)' -o 0,1.2,2.2 phone fax
```

would produce:

```
!Name           Phone Number            Fax Number
Don             +1 123-456-7890         +1 123-456-7899
Hal             +1 234-567-8901         (unknown)
Keith           (unknown)               +1 456-789-0122
Yasushi         +2 345-678-9012         +2 345-678-9011
```

FUTURE DIRECTIONS

The obsolescent **–j** options and the multi-argument **–o** option may be withdrawn from a future issue.

SEE ALSO

awk, *comm*, *sort*, *uniq*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

kill – terminate or signal processes

SYNOPSIS

```
kill -s signal_name pid...
kill -l [exit_status]
OB kill [-signal_name] pid...
OB kill [-signal_name] pid...
```

DESCRIPTION

The *kill* utility will send a signal to the process or processes specified by each *pid* operand.

For each *pid* operand, the *kill* utility will perform actions equivalent to the **XSH** specification *kill*( ) function called with the following arguments:

1. The value of the *pid* operand will be used as the *pid* argument.
2. The *sig* argument is the value specified by the **–s** option, –*signal_number* option, or the –*signal_name* option, or by SIGTERM, if none of these options is specified.

OPTIONS

OB The *kill* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that in the obsolescent form, the –*signal_number* and –*signal_name* options are usually more than a single character.

The following options are supported:

–l (The letter ell.) Write all values of *signal_name* supported by the implementation, if no operand is given. If an *exit_status* operand is given and it is a value of the ? shell special parameter (see Section 2.5.2 on page 27 and *wait*) corresponding to a process that was terminated by a signal, the *signal_name* corresponding to the signal that terminated the process will be written. If an *exit_status* operand is given and it is the unsigned decimal integer value of a signal number, the *signal_name* (the **XSH** specification-defined symbolic constant name without the **SIG** prefix) corresponding to that signal will be written. Otherwise, the results are unspecified.

–s *signal_name*
Specify the signal to send, using one of the symbolic names defined in the **XSH** specification **<signal.h>** description. Values of *signal_name* will be recognised in a case-independent fashion, without the **SIG** prefix. In addition, the symbolic name 0 will be recognised, representing the signal value zero. The corresponding signal will be sent instead of SIGTERM.

OB –*signal_name*
Equivalent to **–s** *signal_name*.

OB *–signal_number*

Specify a non-negative decimal integer, *signal_number*, representing the signal to be used instead of SIGTERM, as the *sig* argument in the effective call to *kill*( ). The correspondence between integer values and the *sig* value used is shown in the following table.

signal_number	*sig* **Value**
0	0
1	SIGHUP
2	SIGINT
3	SIGQUIT
6	SIGABRT
9	SIGKILL
14	SIGALRM
15	SIGTERM

The effects of specifying any *signal_number* other than those listed in the table are undefined.

OB In the obsolescent versions, if the first argument is a negative integer, it will be interpreted as a *–signal_number* option, not as a negative *pid* operand specifying a process group.

OPERANDS

The following operands are supported:

pid One of the following:

1. A decimal integer specifying a process or process group to be signalled. The process or processes selected by positive, negative and zero values of the *pid* operand will be as described for the **XSH** specification *kill*( ) function. If process number 0 is specified, all processes in the process group are signalled. For the effects of negative *pid* numbers, see the **XSH** specification under *kill*( ). If the first *pid* operand is negative, it should be preceded by –– to keep it from being interpreted as an option.
2. A job control job ID (see the **XBD** specification, **Chapter 2**, **Glossary**) that identifies a background process group to be signalled. The job control job ID notation is applicable only for invocations of *kill* in the current shell execution environment; see Section 2.12 on page 63.

   Note: The job control job ID type of *pid* is available on systems supporting both the job control option and the User Portability Utilities Option, which applies to all XSI-conformant systems.

exit_status

A decimal integer specifying a signal number or the exit status of a process terminated by a signal.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *kill*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the **–l** option is not specified, the standard output will not be used.

When the **–l** option is specified, the symbolic name of each signal will be written in the following format:

```
"%s%c", <signal_name>, <separator>
```

where the <*signal_name*> is in upper-case, without the **SIG** prefix, and the <*separator*> will be either a newline character or a space character. For the last signal written, <*separator*> will be a newline character.

When both the **–l** option and *exit_status* operand are specified, the symbolic name of the corresponding signal will be written in the following format:

```
"%s\n", <signal_name>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 At least one matching process was found for each *pid* operand, and the specified signal was successfully processed for at least one matching process.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Process numbers can be found by using *ps*.

The job control job ID notation is not required to work as expected when *kill* is operating in its own utility execution environment. In either of the following examples:

```
nohup kill %1 &
system("kill %1");
```

the *kill* operates in a different environment and will not share the shell's understanding of job numbers.

EXAMPLES

Any of the commands:

```
kill –9 100 -165
kill –s kill 100 -165
kill –s KILL 100 -165
```

sends the SIGKILL signal to the process whose process ID is 100 and to all processes whose process group ID is 165, assuming the sending process has permission to send that signal to the specified processes, and that they exist.

The **XSH** specification and this document do not require specific signal numbers for any *signal_names*. Even the *–signal_number* option provides symbolic (although numeric) names for signals. If a process is terminated by a signal, its exit status indicates the signal that killed it, but the exact values are not specified. The *kill* **–l** option, however, can be used to map decimal signal numbers and exit status values into the name of a signal. The following example reports the status of a terminated job:

```
job
stat=$?
if [ $stat -eq 0 ]
then
       echo job completed successfully.
elif [ $stat -gt 128 ]
then
       echo job terminated by signal SIG$(kill -l $stat).
else
       echo job terminated with error code $stat.
fi
```

To avoid an ambiguity of an initial negative number argument specifying either a signal number or a process group, the ISO / IEC 9945-2: 1993 standard mandates that it always be considered the former. Therefore, to send the default signal to a process group (say 123), an application should use a command similar to one of the following:

```
kill -TERM -123
kill -- -123
```

FUTURE DIRECTIONS

The obsolescent forms may be withdrawn from a future issue. Applications should use the **–s** option.

SEE ALSO

ps, *wait*, the **XSH** specification description of *kill*( ), **<signal.h>**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

lex – generate programs for lexical tasks (**DEVELOPMENT**)

SYNOPSIS

OB `lex -c[-t][ -n| -v][file...]`

DESCRIPTION

The *lex* utility generates C programs to be used in lexical processing of character input, and that can be used as an interface to *yacc*. The C programs are generated from *lex* source code and conform to the ISO C standard. Usually, the *lex* utility writes the program it generates to the file **lex.yy.c**; the state of this file is unspecified if *lex* exits with a non-zero exit status. See **EXTENDED DESCRIPTION** for a complete description of the *lex* input language.

OPTIONS

The *lex* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

OB **–c** Indicate C-language action (default option).

–n Suppress the summary of statistics usually written with the **–v** option. If no table sizes are specified in the *lex* source code and the **–v** option is not specified, then **–n** is implied.

–t Write the resulting program to standard output instead of **lex.yy.c**.

–v Write a summary of *lex* statistics to the standard output. (See the discussion of *lex* table sizes in **Definitions in lex** on page 404.) If the **–t** option is specified and **–n** is not specified, this report will be written to standard error. If table sizes are specified in the *lex* source code, and if the **–n** option is not specified, the **–v** option may be enabled.

OPERANDS

The following operand is supported:

file A pathname of an input file. If more than one such *file* is specified, all files will be concatenated to produce a single *lex* program. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

STDIN

The standard input will be used if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files containing *lex* source code, as described in **EXTENDED DESCRIPTION**.

ENVIRONMENT VARIABLES

If this variable is not set to the POSIX locale, the results are unspecified. The following environment variables affect the execution of *lex*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions. If this variable is not set to the POSIX locale, the results are unspecified.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), and the behaviour of character classes within regular expressions. If this variable is not set to the POSIX locale, the results are unspecified.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If the **–t** option is specified, the text file of C source code output of *lex* will be written to standard output.

If the **–t** option is not specified:

1. Implementation-dependent informational, error and warning messages concerning the contents of *lex* source code input will be written to either the standard output or standard error.
2. If the **–v** option is specified and the **–n** option is not specified, *lex* statistics will also be written to either the standard output or standard error, in an implementation-dependent format. These statistics may also be generated if table sizes are specified with a % operator in the *Definitions* section (see **EXTENDED DESCRIPTION**), as long as the **–n** option is not specified.

STDERR
If the **–t** option is specified, implementation-dependent informational, error and warning messages concerning the contents of *lex* source code input will be written to the standard error.

If the **–t** option is not specified:

1. Implementation-dependent informational, error and warning messages concerning the contents of *lex* source code input will be written to either the standard output or standard error.
2. If the **–v** option is specified and the **–n** option is not specified, *lex* statistics will also be written to either the standard output or standard error, in an implementation-dependent format. These statistics may also be generated if table sizes are specified with a % operator in the *Definitions* section (see **EXTENDED DESCRIPTION**), as long as the **–n** option is not specified.

OUTPUT FILES
A text file containing C source code will be written to **lex.yy.c**, or to the standard output if the **–t** option is present.

EXTENDED DESCRIPTION

Each input file contains *lex* source code, which is a table of regular expressions with corresponding actions in the form of C program fragments.

When **lex.yy.c** is compiled and linked with the *lex* library (using the **–l l** operand with *c89* or *cc*), the resulting program reads character input from the standard input and partitions it into strings that match the given expressions.

When an expression is matched, these actions will occur:

- The input string that was matched is left in *yytext* as a null-terminated string; *yytext* is either an external character array or a pointer to a character string. As explained in **Definitions in lex**, the type can be explicitly selected using the **%array** or **%pointer** declarations, but the default is implementation-dependent.
- The external **int** *yyleng* is set to the length of the matching string.
- The expression's corresponding program fragment, or action, is executed.

During pattern matching, *lex* searches the set of patterns for the single longest possible match. Among rules that match the same number of characters, the rule given first will be chosen.

The general format of *lex* source is:

Definitions
%%
Rules
%%
User Subroutines

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 *Definitions* section beginning with a blank character will be assumed to be a C program fragment and will be copied to the external definition area of the **lex.yy.c** file. Similarly, anything in the *Definitions* section included between delimiter lines containing only **%{** and **%}** will also be copied unchanged to the external definition area of the **lex.yy.c** file.

Any such input (beginning with a blank character or within **%{** and **%}** delimiter lines) appearing at the beginning of the *Rules* section before any rules are specified will be written to **lex.yy.c** after the declarations of variables for the *yylex*( ) function and before the first line of code in *yylex*( ). Thus, user variables local to *yylex*( ) can be declared here, as well as application code to execute upon entry to *yylex*( ).

The action taken by *lex* when encountering any input beginning with a blank character or within **%{** and **%}** delimiter lines appearing in the *Rules* section but coming after one or more rules is undefined. The presence of such input may result in an erroneous definition of the *yylex*( ) function.

Definitions in lex

Definitions appear before the first **%%** delimiter. Any line in this section not contained between **%{** and **%}** lines and not beginning with a blank character is assumed to define a *lex* substitution string. The format of these lines is:

```
name substitute
```

If a *name* does not meet the requirements for identifiers in the ISO C standard, the result is undefined. The string *substitute* will replace the string {*name*} when it is used in a rule. The *name* string is recognised in this context only when the braces are provided and when it does not appear within a bracket expression or within double-quotes.

In the *Definitions* section, any line beginning with a % (percent sign) character and followed by an alphanumeric word beginning with either s or S defines a set of start conditions. Any line beginning with a % followed by a word beginning with either x or X defines a set of exclusive start conditions. When the generated scanner is in a **%s** state, patterns with no state specified will be also active; in a **%x** state, such patterns will not be active. The rest of the line, after the first word, is considered to be one or more blank-character-separated names of start conditions. Start condition names are constructed in the same way as definition names. Start conditions can be used to restrict the matching of regular expressions to one or more states as described in **Regular Expressions in lex** on page 406.

Implementations accept either of the following two mutually exclusive declarations in the *Definitions* section:

%array Declare the type of *yytext* to be a null-terminated character array.

%pointer
Declare the type of *yytext* to be a pointer to a null-terminated character string.

The default type of *yytext* is implementation-dependent. If an application refers to *yytext* outside of the scanner source file (that is, via an **extern**), the application will include the appropriate **%array** or **%pointer** declaration in the scanner source file.

Implementations will accept declarations in the *Definitions* section for setting certain internal table sizes. The declarations are shown in the following table.

Declaration	Description	Minimum Value
%p *n*	Number of positions	2500
%n *n*	Number of states	500
%a *n*	Number of transitions	2000
%e *n*	Number of parse tree nodes	1000
%k *n*	Number of packed character classes	1000
%o *n*	Size of the output array	3000

Table 3-7 Table Size Declarations in *lex*

In the table, *n* represents a positive decimal integer, preceded by one or more blank characters. The exact meaning of these table size numbers is implementation-dependent. The implementation will document how these numbers affect the *lex* utility and how they are related to any output that may be generated by the implementation should space limitations be encountered during the execution of *lex*. It is possible to determine from this output which of the table size values needs to be modified to permit *lex* to successfully generate tables for the input language. The values in the column Minimum Value represent the lowest values conforming implementations will provide.

Rules in lex

The rules in *lex* source files are a table in which the left column contains regular expressions and the right column contains actions (C program fragments) to be executed when the expressions are recognised.

ERE action
ERE action
...

The extended regular expression (*ERE*) portion of a row will be separated from *action* by one or more blank characters. A regular expression containing blank characters is recognised under one of the following conditions:

- The entire expression appears within double-quotes.
- The blank characters appear within double-quotes or square brackets.
- Each blank character is preceded by a backslash character.

User Subroutines in lex

Anything in the user subroutines section will be copied to **lex.yy.c** following *yylex*( ).

Regular Expressions in lex

The *lex* utility supports the set of extended regular expressions (see the **XBD** specification, **Section 7.4**, **Extended Regular Expressions**), with the following additions and exceptions to the syntax:

"**...**" Any string enclosed in double-quotes will represent the characters within the double-quotes as themselves, except that backslash escapes (which appear in the following table) are recognised. Any backslash-escape sequence is terminated by the closing quote. For example, "\01""1" represents a single string: the octal value 1 followed by the character 1.

<state>r
<state1,state2,...>r
The regular expression *r* will be matched only when the program is in one of the start conditions indicated by *state*, *state1* and so forth; see **Actions in lex** on page 409. (As an exception to the typographical conventions of the rest of this document, in this case *<state>* does not represent a metavariable, but the literal angle-bracket characters surrounding a symbol.) The start condition is recognised as such only at the beginning of a regular expression.

r/*x* The regular expression *r* will be matched only if it is followed by an occurrence of regular expression *x*. The token returned in *yytext* will only match *r*. If the trailing portion of *r* matches the beginning of *x*, the result is unspecified. The *r* expression cannot include further trailing context or the $ (match-end-of-line) operator; *x* cannot include the ^ (match-beginning-of-line) operator, nor trailing context, nor the $ operator. That is, only one occurrence of trailing context is allowed in a *lex* regular expression, and the ^ operator only can be used at the beginning of such an expression.

{*name***}** When *name* is one of the substitution symbols from the *Definitions* section, the string, including the enclosing braces, will be replaced by the *substitute* value. The *substitute* value will be treated in the extended regular expression as if it were enclosed in parentheses. No substitution will occur if **{***name***}** occurs within a bracket expression or within double-quotes.

Within an ERE, a backslash character is considered to begin an escape sequence as specified in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v). In addition, the escape sequences in the following table will be recognised.

A literal newline character cannot occur within an ERE; the escape sequence \n can be used to represent a newline character. A newline character cannot be matched by a period operator.

Escape Sequence	Description	Meaning
digits	A backslash character followed by the longest sequence of one, two or three octal-digit characters (01234567). If all of the digits are 0, (that is, representation of the NUL character), the behaviour is undefined.	The character whose encoding is represented by the one-, two- or three-digit octal integer. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-dependent. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
x*digits*	A backslash character followed by the longest sequence of hexadecimal-digit characters (01234567abcdefABCDEF). If all of the digits are 0, (that is, representation of the NUL character), the behaviour is undefined.	The character whose encoding is represented by the hexadecimal integer.
c	A backslash character followed by any character not described in this table or in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v).	The character *c*, unchanged.

Table 3-8 Escape Sequences in *lex*

The order of precedence given to extended regular expressions for *lex* differs from that specified in the **XBD** specification, **Section 7.4**, **Extended Regular Expressions**. The order of precedence for *lex* is as shown in the following table, from high to low.

Note: The escaped characters entry is not meant to imply that these are operators, but they are included in the table to show their relationships to the true operators. The start condition, trailing context and anchoring notations have been omitted from the table because of the placement restrictions described in this section; they can only appear at the beginning or ending of an ERE.

collation-related bracket symbols	**[= =] [: :] [. .]**
escaped characters	\<*special character*>
bracket expression	**[ ]**
quoting	**" ..."**
grouping	**( )**
definition	**{***name***}**
single-character RE duplication	*** + ?**
concatenation	
interval expression	**{***m***,***n***}**
alternation	**|**

Table 3-9 ERE Precedence in *lex*

The ERE anchoring operators (^ and $) do not appear in the table. With *lex* regular expressions, these operators are restricted in their use: the ^ operator can only be used at the beginning of an entire regular expression, and the $ operator only at the end. The operators apply to the entire regular expression. Thus, for example, the pattern (^abc)|(def$) is undefined; it can instead be written as two separate rules, one with the regular expression ^abc and one with def$, which share a common action via the special | action (see below). If the pattern were written ^abc|def$, it would match either of **abc** or **def** on a line by itself.

Unlike the general ERE rules, embedded anchoring is not allowed by most historical *lex* implementations. An example of embedded anchoring would be for patterns such as (^|)foo(|$) to match **foo** when it exists as a complete word. This functionality can be obtained using existing *lex* features:

```
^foo/[ \n]      |
" foo"/[ \n]    /* found foo as a separate word */
```

Note also that $ is a form of trailing context (it is equivalent to /\n) and as such cannot be used with regular expressions containing another instance of the operator (see the preceding discussion of trailing context).

The additional regular expressions trailing-context operator / can be used as an ordinary character if presented within double-quotes, "/"; preceded by a backslash, \/; or within a bracket expression, **[/]**. The start-condition < and > operators are special only in a start condition at the beginning of a regular expression; elsewhere in the regular expression they are treated as ordinary characters.

The following examples clarify the differences between *lex* regular expressions and regular expressions appearing elsewhere in this document. For regular expressions of the form *r*/*x*, the string matching *r* is always returned; confusion may arise when the beginning of *x* matches the trailing portion of *r*. For example, given the regular expression a*b/cc and the input **aaabcc**, *yytext* would contain the string **aaab** on this match. But given the regular expression x*/xy and the input **xxxy**, the token **xxx**, not **xx**, is returned by some implementations because **xxx** matches x*.

In the rule ab*/bc, the b* at the end of *r* will extend *r*'s match into the beginning of the trailing context, so the result is unspecified. If this rule were ab/bc, however, the rule matches the text **ab** when it is followed by the text **bc**. In this latter case, the matching of *r* cannot extend into the beginning of *x*, so the result is specified.

Actions in lex

The action to be taken when an *ERE* is matched can be a C program fragment or the special actions described below; the program fragment can contain one or more C statements, and can also include special actions. The empty C statement ; is a valid action; any string in the **lex.yy.c** input that matches the pattern portion of such a rule is effectively ignored or skipped. However, the absence of an action is not valid, and the action *lex* takes in such a condition is undefined.

The specification for an action, including C statements and special actions, can extend across several lines if enclosed in braces:

```
ERE <one or more blanks> { program statement
                           program statement }
```

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 behaviour 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 **%%** generates a C program that simply copies the input to the output unchanged.

Four special actions are available:

```
|   ECHO;   REJECT;   BEGIN
```

`|` The action | means that the action for the next rule is the action for this rule. Unlike the other three actions, | cannot be enclosed in braces or be semicolon-terminated; it must be specified alone, with no other actions.

`ECHO;` Write the contents of the string *yytext* on the output.

`REJECT;`
Usually 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 the second choice after the current rule to be executed for the same input. Thus, multiple rules can be matched and executed for one input string or overlapping input strings. For example, given the regular expressions **xyz** and **xy** and the input **xyz**, usually only the regular expression **xyz** would match. The next attempted match would start after z. If the last action in the **xyz** rule is **REJECT**, both this rule and the **xy** rule would be executed. The **REJECT** action may be implemented in such a fashion that flow of control does not continue after it, as if it were equivalent to a **goto** to another part of *yylex*( ). The use of **REJECT** may result in somewhat larger and slower scanners.

`BEGIN` The action:

```
BEGIN newstate;
```

switches the state (start condition) to *newstate*. If the string *newstate* has not been declared previously as a start condition in the *Definitions* section, the results are unspecified. The initial state is indicated by the digit 0 or the token **INITIAL**.

The functions or macros described below are accessible to user code included in the *lex* input. It is unspecified whether they appear in the C code output of *lex*, or are accessible only through the **–l l** operand to *c89* or *cc* (the *lex* library).

`int yylex(void)`
Performs lexical analysis on the input; this is the primary function generated by the **lex** utility. The function returns zero when the end of input is reached; otherwise it returns non-zero values (tokens) determined by the actions that are selected.

`int yymore(void)`
: When called, indicates that when the next input string is recognised, it is to be appended to the current value of *yytext* rather than replacing it; the value in *yyleng* is adjusted accordingly.

`int yyless(int n)`
: Retains *n* initial characters in *yytext*, NUL-terminated, and treats the remaining characters as if they had not been read; the value in *yyleng* is adjusted accordingly.

`int input(void)`
: Returns the next character from the input, or zero on end-of-file. It obtains input from the stream pointer *yyin*, although possibly via an intermediate buffer. Thus, once scanning has begun, the effect of altering the value of *yyin* is undefined. The character read is removed from the input stream of the scanner without any processing by the scanner.

`int unput(int c)`
: Returns the character *c* to the input; *yytext* and *yyleng* are undefined until the next expression is matched. The result of using *unput* for more characters than have been input is unspecified.

The following functions appear only in the *lex* library accessible through the **–l l** operand; they can therefore be redefined by a portable application:

`int yywrap(void)`
: Called by *yylex*( ) at end-of-file; the default *yywrap*( ) always will return 1. If the application requires *yylex*( ) to continue processing with another source of input, then the application can include a function *yywrap*( ), which associates another file with the external variable **FILE** **yyin* and will return a value of zero.

`int main(int argc, char *argv[])`
: Calls *yylex*( ) to perform lexical analysis, then exits. The user code can contain *main*( ) to perform application-specific operations, calling *yylex*( ) as applicable.

The reason for breaking these functions into two lists is that only those functions in **libl.a** can be reliably redefined by a portable application.

Except for *input*( ), *unput*( ) and *main*( ), all external and static names generated by *lex* begin with the prefix **yy** or **YY**.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Portable applications are warned that in the *Rules* section, an *ERE* without an action is not acceptable, but need not be detected as erroneous by *lex*. This may result in compilation or run-time errors.

The purpose of *input*( ) is to take characters off the input stream and discard them as far as the lexical analysis is concerned. A common use is to discard the body of a comment once the beginning of a comment is recognised.

The *lex* utility is not fully internationalised in its treatment of regular expressions in the *lex* source code or generated lexical analyser. It would seem desirable to have the lexical analyser interpret the regular expressions given in the *lex* source according to the environment specified when the lexical analyser is executed, but this is not possible with the current *lex* technology. Furthermore, the very nature of the lexical analysers produced by *lex* must be closely tied to the lexical requirements of the input language being described, which will frequently be locale-specific anyway. (For example, writing an analyser that is used for French text will not automatically be useful for processing other languages.)

EXAMPLES

The following is an example of a *lex* program that implements a rudimentary scanner for a Pascal-like syntax:

```
%{
/* need this for the call to atof() below */
#include <math.h>
/* need this for printf(), fopen() and stdin below */
#include <stdio.h>
%}
DIGIT    [0-9]
ID       [a-z][a-z0-9]*
%%
{DIGIT}+    {
       printf("An integer: %s (%d)\n", yytext,
              atoi(yytext));
       }
{DIGIT}+"."{DIGIT}*        {
       printf("A float: %s (%g)\n", yytext,
              atof(yytext));
       }
if|then|begin|end|procedure|function        {
       printf("A keyword: %s\n", yytext);
       }
{ID}   printf("An identifier: %s\n", yytext);
"+"|"-"|"*"|"/"            printf("An operator: %s\n", yytext);
"{"[^}\n]*"}"       /* eat up one-line comments */
[ \t\n]+            /* eat up white space */
.      printf("Unrecognised character: %s\n", yytext);
%%
int main(int argc, char *argv[])
{
       ++argv, --argc;  /* skip over program name */
       if (argc > 0)
              yyin = fopen(argv[0], "r");
       else
              yyin = stdin;
       yylex();
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

c89, *yacc*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

line – read one line (**TO BE WITHDRAWN**)

SYNOPSIS

EX `line`

DESCRIPTION

The *line* utility copies one line (up to and including a newline) from the standard input and writes it to standard output. It always writes at least a newline.

OPTIONS

None.

OPERANDS

None.

STDIN

The standard input is a text file.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file consisting of one line.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Exit status is:

0 Successful completion.
>0 End-of-file on input.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *line* utility is often used within command scripts to read from the user's terminal.

SEE ALSO

read.

EXAMPLES

None.

FUTURE DIRECTIONS

The *line* utility has not been updated for internationalisation features. It will be withdrawn from a future issue. The *read* utility should be used instead.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Marked **TO BE WITHDRAWN**.

NAME

lint – check C-language programs (**DEVELOPMENT**) (**TO BE WITHDRAWN**)

SYNOPSIS

EX

```
lint [-abcnpuxv] [-D name=value]][-I directory][-L directory]
[-o x][-U name] operand...
```

DESCRIPTION

The *lint* utility cross-checks multiple C-language source files and library definitions and reports potential errors. Among the error conditions that are detected are:

- unreachable statements
- loops not entered at the top
- automatic variables declared and not used
- inconsistent declarations between files
- non-portable constructions
- logical expressions whose value is constant
- functions that return values in some places and not in others
- functions called with varying numbers or types of arguments
- functions whose values are not used or whose values are used when none are returned.

The *lint* utility takes all the files with .c and .ln suffixes, and any additional lint libraries specified by the **–l** operand, and processes 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**), if it exists, will be appended instead. When the **–c** option is not used, *lint* checks this list of files for mutual compatibility. When the **–c** option is used, the .ln files and the lint libraries are ignored.

Certain conventional comments in the C source change the behaviour of *lint*:

`/*NOTREACHED*/` Stop comments about unreachable code at appropriate points.

`/*VARARGS`*n*`*/` Suppress 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 zero.

`/*ARGSUSED*/` Suppress diagnostic messages about unused arguments in the next function.

`/*LINTLIBRARY*/` Suppress, at the beginning of a file, diagnostic messages about unused functions and function arguments in this file. This is equivalent to using the **–v** and **–x** options.

Other comments in the C source of the form:

```
/\*[[:upper:]][[:upper:][:digit:]]*\*/
```

(where the form is expressed using the syntax of basic regular expressions, defined in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**) may be interpreted by *lint* in implementation-dependent ways.

If the **–c** option is specified, then for all pathname operands of the form *file*.c the files:

```
$(basename pathname .c).ln
```

are produced.

If the **–o** option is present with option-argument *x*, a file with the name:

```
llib-l$(basename x).ln
```

is produced.

The *lint* utility produces its first output on a per-source-file basis. Diagnostic messages regarding input 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 diagnostic message stems from a given source file or from one of its included files, the source filename will be printed followed by a question mark.

During the execution of *lint*, values are established for certain predefined macros from the ISO C standard: __LINE__, __FILE__, __DATE__, __TIME__ and __STDC__.

OPTIONS

The *lint* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the **–l** operands have the format of options, but their position within a list of operands affects the order in which libraries are searched.

The following options are supported:

–a Suppress diagnostic messages about assignments of long values to variables that are not long.

–b Suppress diagnostic messages about **break** statements that cannot be reached.

–c Produce a **.ln** file for every **.c** file on the command line. These **.ln** files are not checked for interfunction compatibility.

–h Do not apply heuristic tests that attempt to diagnose bugs intuitively, improve style and reduce waste.

–n Do not check compatibility against either the standard or the portable lint library.

–o *x* Produce a lint library with the name:

```
llib-l$(basename x).ln.
```

The **–c** option nullifies any use of the **–o** option. The **–o** option causes this file to be saved in the named lint library. To produce the lint library without extraneous messages, the **–x** option should be used. The **–v** option is useful if the source files for the lint library are just external interfaces. These option settings are also available through the use of the ‘‘lint comments’’ listed in the **DESCRIPTION**.

–p Cause all non-external names to be treated as if they were truncated to thirty-one characters and all external names truncated to six characters, folded to one case. Append the portable C lint library, if it exists, to the end of the list of files.

–u Suppress diagnostic messages 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 diagnostic messages about unused arguments in functions.

–x Do not report variables referred to by external declarations but never used.

The **–D**, **–U**, **–L** and **–I** options of the C compiler (see *cc* and *c89*) are also recognised as separate arguments.

The **–g** and **–O** options of the C compiler are also recognised as separate arguments, but are ignored. (By recognising these options, the behaviour of *lint* is closer to that of the *cc* utility.) Other options are ignored, and a warning message may be issued. The pre-defined macro lint (for common-usage C) or __LINT__ (for the ISO C standard) is defined to allow certain questionable code to be altered or removed for *lint*.

OPERANDS

The following operands are supported:

*file***.c** A pathname naming a C-language source file.

*file***.ln** A pathname of a file analogous to a **.o** file produced by the C compiler.

An operand of the form **–l** *x* means search the library named **llib-l***x***.ln** or **llib-l***x* if **llib-l***x***.ln** is not readable.

The processing of other files is implementation-dependent.

The *lint* utility will recognise the following **–l** operands for standard libraries:

–l m Names the library **llib-lm.ln**, which contains the functions described by the ISO C standard with prototypes in the **<math.h>** header.

–l c Names the Standard C library, **llib-lc.ln**, which will contain everything else defined by the ISO C standard. The library searched also will include all functions defined by the **XSH** specification. This operand is not required to be present to cause a search of the Standard C library.

–l l Names the library **llib-ll.ln**, which will contain functions required by the C-language output of *lex* that are not available through the Standard C library.

–l y Names the library **llib-ly.ln**, which contains functions required by the C-language output of *yacc* that are not available through the Standard C library.

It is unspecified whether the libraries **llib-lm.ln**, **llib-lc.ln**, **llib-ll.ln** or **llib-ly.ln** (or any other library accessed via **–l**) exist as regular files.

STDIN

Not used.

INPUT FILES

The input file must be one of the following: a **.c** suffixed text file containing C-language source or a **.ln** suffixed file of unspecified format produced by a previous invocation of *lint* with a **–c** or **–o** option.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *lint*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The format of the report produced by *lint* is unspecified.

STDERR
Used only for diagnostic and warning messages.

OUTPUT FILES
The format of **.ln** files is unspecified.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The behaviour 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 that corresponds to the **.c** file, and prints all messages that pertain to 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 **–l** *x* options. This will print all the interfile 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*. If used incrementally on a set of files, the result of partial checks can be saved to speed subsequent intermodule consistency checks.

The lint library often contains all routines in that library stripped of everything but prototypes and return values.

The intent of allowing unrecognised suffixes is to permit implementations to recognise things like archives and source of other languages, but not require all implementations to do so. Portable applications should use only the suffixes described in this document.

Programs produced by *lex* or *yacc* will often result in many diagnostic messages about **break** statements that cannot be reached. Use of **–b** is recommended.

EXAMPLES

Assuming a file hierarchy as shown here:

```
$ ls -R
 libsrc:
applib1.c
applib2.c
applib3.c
applib.a
 appsrc:
app1.c
app2.c
$ cd libsrc
```

The following creates **llib-lapplib.ln** for later use:

```
$ lint -o applib applib*.c
$ cd ../appsrc
```

The following checks the source for both applications against the previously created library:

```
$ for app in *.c
> do
>        lint -L ../libsrc $app -l applib
> done
```

FUTURE DIRECTIONS

The *lint* utility will be withdrawn from a future issue.

SEE ALSO

cc, *c89*, *lex*, *make*, *yacc*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support made optional.

NAME

ln – link files

SYNOPSIS

```
ln [-f] source_file target_file
ln [-f] source_file... target_dir
```

DESCRIPTION

In the first synopsis form, the *ln* utility will create a new directory entry (link) for the file specified by the *source_file* operand, at the *destination* path specified by the *target_file* operand. This first synopsis form is assumed when the final operand does not name an existing directory; if more than two operands are specified and the final is not an existing directory, an error will result.

In the second synopsis form, the *ln* utility will create a new directory entry for each file specified by a *source_file* operand, at a *destination* path in the existing directory named by *target_dir*.

If the last operand specifies an existing file of a type not specified by the **XSH** specification, the behaviour is implementation-dependent.

The corresponding destination path for each *source_file* will be the concatenation of the target directory pathname, a slash character, and the last pathname component of the *source_file*. The second synopsis form will be assumed when the final operand names an existing directory.

For each *source_file*:

1. If the *destination* path exists:
   a. If the **–f** option is not specified, *ln* will write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_files*.
   b. Actions will be performed equivalent to the **XSH** specification *unlink*( ) function, called using *destination* as the *path* argument. If this fails for any reason, *ln* will write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_files*.
2. Actions will be performed equivalent to the **XSH** specification *link*( ) function using *source_file* as the *path1* argument, and the *destination* path as the *path2* argument.

OPTIONS

The *ln* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–f Force existing *destination* pathnames to be removed to allow the link.

OPERANDS

The following operands are supported:

source_file
A pathname of a file to be linked. This can be a regular or special file; whether a directory can be linked is implementation-dependent.

target_file
The pathname of the new directory entry to be created.

target_dir
A pathname of an existing directory in which the new directory entries are to be created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ln*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All the specified files were linked successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, find, pax, rm.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

locale – get locale-specific information

SYNOPSIS

```
locale [ -a| -m]

locale [-ck] name...
```

DESCRIPTION

The *locale* utility writes information about the current locale environment, or all public locales, to the standard output. For the purposes of this section, a *public locale* is one provided by the implementation that is accessible to the application.

When *locale* is invoked without any arguments, it summarises the current locale environment for each locale category as determined by the settings of the environment variables defined in the **XBD** specification, **Chapter 5**, **Locale**.

When invoked with operands, it writes values that have been assigned to the keywords in the locale categories, as follows:

- Specifying a keyword name selects the named keyword and the category containing that keyword.
- Specifying a category name selects the named category and all keywords in that category.

OPTIONS

The *locale* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Write information about all available public locales. The available locales include **POSIX**, representing the POSIX locale. The manner in which the implementation determines what other locales are available is implementation-dependent.

–c Write the names of selected locale categories; see **STDOUT**. The **–c** option increases readability when more than one category is selected (for example, via more than one keyword name or via a category name). It is valid both with and without the **–k** option.

–k Write the names and values of selected keywords. The implementation may omit values for some keywords; see **OPERANDS**.

–m Write names of available charmaps; see the **XBD** specification, **Section 4.1**, **Portable Character Set**.

OPERANDS

The following operand is supported:

name The name of a locale category as defined in the **XBD** specification, **Chapter 5**, **Locale**, the name of a keyword in a locale category, or the reserved name **charmap**. The named category or keyword will be selected for output. If a single *name* represents both a locale category name and a keyword name in the current locale, the results are unspecified. Otherwise, both category and keyword names can be specified as *name* operands, in any sequence. It is implementation-dependent whether any keyword values are written for the categories LC_CTYPE and LC_COLLATE.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *locale*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

EX The *LANG*, *LC_** and *NLSPATH* environment variables must specify the current locale environment to be written out; they will be used if the **–a** option is not specified.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If *locale* is invoked without any options or operands, the names and values of the *LANG* and *LC_** environment variables described in this document will be written to the standard output, one variable per line, with *LANG* first, and each line using the following format. Only those variables set in the environment and not overridden by *LC_ALL* will be written using this format:

```
"%s=%s\n", <variable_name>, <value>
```

The names of those *LC_** variables associated with locale categories defined in this document that are not set in the environment or are overridden by *LC_ALL* will be written in the following format:

```
"%s=\"%s\"\n", <variable_name>, <implied value>
```

The <*implied value*> is the name of the locale that has been selected for that category by the implementation, based on the values in *LANG* and *LC_ALL*, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

The <*value*> and <*implied value*> shown above will be properly quoted for possible later reentry to the shell. The <*value*> will not be quoted using double-quotes (so that it can be distinguished by the user from the <*implied value*> case, which always requires double-quotes).

The *LC_ALL* variable will be written last, using the first format shown above. If it is not set, it will be written as:

```
"LC_ALL=\n"
```

If any arguments are specified:

1. If the **–a** option is specified, the names of all the public locales will be written, each in the following format:

   "%s\n", <*locale name*>

2. If the **–c** option is specified, the names of all selected categories will be written, each in the following format:

   "%s\n", <*category name*>

   If keywords are also selected for writing (see following items), the category name output will precede the keyword output for that category.

   If the **–c** option is not specified, the names of the categories will not be written; only the keywords, as selected by the *name* operand, will be written.

3. If the **–k** option is specified, the names and values of selected keywords will be written. If a value is non-numeric, it will be written in the following format:

   "%s=\"%s\"\n", <*keyword name*>, <*keyword value*>

   If the keyword was **charmap**, the name of the charmap (if any) that was specified via the *localedef* **–f** option when the locale was created will be written, with the word **charmap** as <*keyword name*>.

   If a value is numeric, it will be written in one of the following formats:

   "%s=%d\n", <*keyword name*>, <*keyword value*>

   "%s=%c%o\n", <*keyword name*>, <*escape character*>, <*keyword value*>

   "%s=%cx%x\n", <*keyword name*>, <*escape character*>,
   <*keyword value*>

   where the <*escape character*> is that identified by the **escape_char** keyword in the current locale; see the **XBD** specification, **Section 5.3**, **Locale Definition**.

   Compound keyword values (list entries) will be separated in the output by semicolons. When included in keyword values, the semicolon, the double-quote, the backslash and any control character will be preceded (escaped) with the escape character.

4. If the **–k** option is not specified, selected keyword values will be written, each in the following format:

   "%s\n", <*keyword value*>

   If the keyword was **charmap**, the name of the charmap (if any) that was specified via the *localedef* **–f** option when the locale was created will be written.

5. If the **–m** option is specified, then a list of all available charmaps will be written, each in the format:

   "%s\n", <*charmap*>

   where <*charmap*> is in a format suitable for use as the option-argument to the *localedef* **–f** option.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All the requested information was found and output successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If the *LANG* environment variable is not set or set to an empty value, or one of the *LC_** environment variables is set to an unrecognised value, the actual locales assumed (if any) are implementation-dependent as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

Implementations are not required to write out the actual values for keywords in the categories LC_CTYPE and LC_COLLATE; however, they must write out the categories (allowing an application to determine, for example, which character classes are available).

EXAMPLES

In the following examples, the assumption is that locale environment variables are set as follows:

```
LANG=locale_x
LC_COLLATE=locale_y
```

The command:

```
locale
```

would result in the following output:

```
LANG=locale_x
LC_CTYPE="locale_x"
LC_COLLATE=locale_y
LC_TIME="locale_x"
LC_NUMERIC="locale_x"
LC_MONETARY="locale_x"
LC_MESSAGES="locale_x"
LC_ALL=
```

The order of presentation of the categories is not specified by this document.

The command:

```
LC_ALL=POSIX locale -ck decimal_point
```

would produce:

```
LC_NUMERIC
decimal_point="."
```

The following command shows an application of *locale* to determine whether a user-supplied response is affirmative:

```
if printf "%s\n" "$response" | grep -Eq "$(locale yesexpr)"
then
       affirmative processing goes here
else
       non-affirmative processing goes here
fi
```

FUTURE DIRECTIONS

None.

SEE ALSO

localedef, the **XBD** specification, **Section 5.3**, **Locale Definition**.

CHANGE HISTORY

First released in Issue 4.

NAME

localedef – define locale environment

SYNOPSIS

`localedef [-c][-f` *charmap*`][-i` *sourcefile*`]` *name*

DESCRIPTION

The *localedef* utility converts source definitions for locale categories into a format usable by the functions and utilities whose operational behaviour is determined by the setting of the locale environment variables defined in the **XBD** specification, **Chapter 5**, **Locale**. It is implementation-dependent whether users have the capability to create new locales, in addition to those supplied by the implementation. Since the symbolic constant {POSIX2_LOCALEDEF} is defined on all XSI-conformant systems, the system supports the creation of new locales.

EX

The utility reads source definitions for one or more locale categories belonging to the same locale from the file named in the **–i** option (if specified) or from standard input.

The *name* operand identifies the target locale. The utility supports the creation of *public*, or generally accessible locales, as well as *private*, or restricted-access locales. Implementations may restrict the capability to create or modify public locales to users with the appropriate privileges.

Each category source definition is identified by the corresponding environment variable name and terminated by an **END** *category-name* statement. The following categories are supported. In addition, the input may contain source for implementation-dependent categories.

LC_CTYPE
: Defines character classification and case conversion.

LC_COLLATE
: Defines collation rules.

LC_MONETARY
: Defines the format and symbols used in formatting of monetary information.

LC_NUMERIC
: Defines the decimal delimiter, grouping and grouping symbol for non-monetary numeric editing.

LC_TIME
: Defines the format and content of date and time information.

LC_MESSAGES
: Defines the format and values of affirmative and negative responses.

OPTIONS

The *localedef* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c Create permanent output even if warning messages have been issued.

–f *charmap*
: Specify the pathname of a file containing a mapping of character symbols and collating element symbols to actual character encodings. The format of the *charmap* is described under the **XBD** specification, **Section 4.4**, **Character Set Description File**. This option must be specified if symbolic names (other than collating symbols defined in a **collating-symbol** keyword) are used. If the **–f** option is not present, an implementation-dependent character mapping will be used.

–i *inputfile*
The pathname of a file containing the source definitions. If this option is not present, source definitions will be read from standard input. The format of the *inputfile* is described in the **XBD** specification, **Section 5.3**, **Locale Definition**.

OPERANDS

The following operand is supported:

name Identifies the locale. See the **XBD** specification, **Chapter 5**, **Locale** for a description of the use of this name. If the name contains one or more slash characters, *name* will be interpreted as a pathname where the created locale definitions will be stored. If *name* does not contain any slash characters, the interpretation of the name is implementation-dependent and the locale will be public. This capability may be restricted to users with appropriate privileges. (As a consequence of specifying one *name*, although several categories can be processed in one execution, only categories belonging to the same locale can be processed.)

STDIN

Unless the **–i** option is specified, the standard input must be a text file containing one or more locale category source definitions, as described in the **XBD** specification, **Section 5.3**, **Locale Definition**. When lines are continued using the escape character mechanism, there is no limit to the length of the accumulated continued line.

INPUT FILES

The character set mapping file specified as the *charmap* option-argument is described under the **XBD** specification, **Section 4.4**, **Character Set Description File**. If a locale category source definition contains a **copy** statement, as defined in the **XBD** specification, **Chapter 5**, **Locale**, and the **copy** statement names a valid, existing locale, then *localedef* will behave as if the source definition had contained a valid category source definition for the named locale.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *localedef*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
(This variable has no affect on *localedef*; the POSIX locale will be used for this category.)

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files). This variable has no affect on the processing of *localedef* input data; the POSIX locale is used for this purpose, regardless of the value of this variable.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The utility will report all categories successfully processed, in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The format of the created output is unspecified. If the *name* operand does not contain a slash, the existence of an output file for the locale is unspecified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 No errors occurred and the locales were successfully created.
1 Warnings occurred and the locales were successfully created.
2 The locale specification exceeded implementation limits or the coded character set or sets used were not supported by the implementation, and no locale was created.
3 The capability to create new locales is not supported by the implementation.
>3 Warnings or errors occurred and no output was created.

CONSEQUENCES OF ERRORS

If an error is detected, no permanent output will be created.

If warnings occur, permanent output will be created if the **–c** option was specified. The following conditions will cause warning messages to be issued:

- If a symbolic name not found in the *charmap* file is used for the descriptions of the LC_CTYPE or LC_COLLATE categories (for other categories, this will be an error conditions).
- If the number of operands to the **order** keyword exceeds the {COLL_WEIGHTS_MAX} limit.
- If optional keywords not supported by the implementation are present in the source.

Other implementation-dependent conditions may also cause warnings.

APPLICATION USAGE

The *charmap* definition is optional, and is contained outside the locale definition. This allows both completely self-defined source files, and generic sources (applicable to more than one codeset). To aid portability, all *charmap* definitions must use the same symbolic names for the portable character set. As explained in the **XBD** specification, **Section 4.4**, **Character Set Description File**, it is implementation-dependent whether or not users or applications can provide additional character set description files. Therefore, the **–f** option might be operable only when an implementation-provided *charmap* is named.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

locale, the **XBD** specification, **Section 5.3**, **Locale Definition**.

CHANGE HISTORY

First released in Issue 4.

NAME

logger – log messages

SYNOPSIS

`logger` *`string`*

DESCRIPTION

The *logger* utility saves a message, in an unspecified manner and format, containing the *string* operands provided by the user. The messages are expected to be evaluated later by personnel performing system administration tasks.

It is implementation-dependent whether messages written in locales other than the POSIX locale are effective.

OPTIONS

None.

OPERANDS

The following operand is supported:

string One of the string arguments whose contents are concatenated together, in the order specified, separated by single space characters.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *logger*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error. (This means diagnostics from *logger* to the user or application, not diagnostic messages that the user is sending to the system administrator.)

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Unspecified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility allows logging of information for later use by a system administrator or programmer in determining why non-interactive utilities have failed. The locations of the saved messages, their format and retention period are all unspecified. There is no method for a portable application to read messages, once written.

EXAMPLES

A batch application, running non-interactively, tries to read a configuration file and fails; it may attempt to notify the system administrator with:

```
logger myname: unable to read file foo. [timestamp]
```

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *write*.

CHANGE HISTORY

First released in Issue 4.

NAME

logname – return user's login name

SYNOPSIS

```
logname
```

DESCRIPTION

The *logname* utility will write the user's login name to standard output. The login name is the string that would be returned by the **XSH** specification *getlogin*( ) function. Under the conditions where the *getlogin*( ) function would fail, the *logname* utility will write a diagnostic message to standard error and exit with a non-zero exit status.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *logname*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *logname* utility output will be a single line consisting of the user's login name:

```
"%s\n", <login name>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *logname* utility explicitly ignores the *LOGNAME* environment variable because environment changes could produce erroneous results.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

id, *who*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

lp – send files to a printer

SYNOPSIS

```
lp [-c][-d dest][-n copies][-msw][-o option]... [-t title]
[file...]
```

DESCRIPTION

The *lp* utility copies the input files to an output device in an unspecified manner. The default output destination should be to a hardcopy device, such as a line printer or microfilm recorder, that produces non-volatile, human-readable documents. If such a device is not available to the application, or if the system provides no such device, the *lp* utility will exit with a non-zero exit status.

The actual writing to the output device may occur some time after the *lp* utility successfully exits. During the portion of the writing that corresponds to each input file, the implementation guarantees exclusive access to the device.

EX Normally, a banner page is produced to separate and identify each print job. This page may be suppressed by implementation-dependent conditions, such as an operator command or one of the **–o** *option* values.

OPTIONS

The *lp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c Exit only after further access to any of the input files is no longer required. The application can then safely delete or modify the files without affecting the output operation. 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. On some systems, **–c** may be on by default.

–d *dest*

EX Specify a string that names the output device or destination. If *dest* is a printer, the request will be printed only on that specific printer. If *dest* is a class of printers, 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, and so forth), requests for specific destinations need not be accepted; see *lpstat*. Destination names vary between systems; see *lpstat*.

If **–d** is not specified, and neither the *LPDEST* nor *PRINTER* environment variable is set, an unspecified output device is used. The **–d** *dest* option takes precedence over *LPDEST*, which in turn takes precedence over *PRINTER*. Results are undefined when *dest* contains a value that is not a valid device or destination name.

EX UN **–m** Send mail (see *mailx*) after the files have been printed. By default, no mail is sent upon normal completion of the print request.

–n *copies*

Write *copies* number of copies of the files, where *copies* is a positive decimal integer. The methods for producing multiple copies and for arranging the multiple copies when multiple *file* operands are used are unspecified, except that each file will be output as an integral whole, not interleaved with portions of other files.

EX UN **–o** *option*
Specify printer-dependent or class-dependent *option*s. Several such *option*s may be collected by specifying the **–o** option more than once.

EX PI **–s** Suppress messages from *lp* such as ‘‘request id is …’’.

EX UN **–t** *title* Write *title* on the banner page of the output.

EX UN **–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.

OPERANDS

The following operand is supported:

file A pathname of a file to be output. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used. If a *file* operand is used, but the **–c** option is not specified, the process performing the writing to the output device may have user and group permissions that differ from that of the process invoking *lp*.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *lp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *LC_TIME*
Determine the format and contents of date and time strings displayed in the *lp* banner page, if any.

LPDEST
Determine the output device or destination. If the *LPDEST* environment variable is not set, the *PRINTER* environment variable will be used. The **–d** *dest* option takes precedence over *LPDEST*. Results are undefined when **–d** is not specified and *LPDEST* contains a value that is not a valid device or destination name.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PRINTER

Determine the output device or destination. If the *LPDEST* and *PRINTER* environment variables are not set, an unspecified output device is used. The **–d** *dest* option and the *LPDEST* environment variable takes precedence over *PRINTER*. Results are undefined when **–d** is not specified, *LPDEST* is unset, and *PRINTER* contains a value that is not a valid device or destination name.

ASYNCHRONOUS EVENTS

Default.

STDOUT

EX The *lp* utility associates a unique *ID* with each request and writes it to the standard output, unless **–s** is specified. The format of the message is unspecified. This *ID* can be used later to cancel (see *cancel*) or find the status (see *lpstat*) of the request.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All input files were processed successfully.
>0 No output device was available, or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *pr* and *fold* utilities can be used to achieve reasonable formatting for the implementation's default page size.

A portable application can use one of the *file* operands only with the **–c** option or if the file is publicly readable and guaranteed to be available at the time of printing. This is because the standard gives the implementation the freedom to queue up the request for printing at some later time by a different process that might not be able to access the file.

EXAMPLES

1. To print file *file*:

```
lp –c file
```

2. To print multiple files with headers:

```
pr file1 file2 | lp
```

FUTURE DIRECTIONS

None.

SEE ALSO

banner, *cancel*, *lpstat*, *mailx*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

lpstat – report line printer status information

SYNOPSIS

UN EX

```
lpstat [-drst] [-a[list]] [-c[list]] [-o[list]] [-p[list]] [-u[lst]]
[-v[list]] [ID...]
```

DESCRIPTION

The *lpstat* utility writes to standard output information about the current status of the line printer system.

If no arguments are given, *lpstat* writes the status of all requests made to *lp* by the user that are still in the output queue.

OPTIONS

The *lpstat* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except the option-arguments are optional and cannot be presented as separate arguments.

Some of the options below can 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 quoted list of items separated from
OB one another by a comma or one or more blank characters, or combinations of both. See **EXAMPLES**.

The omission of a *list* following such options causes all information relevant to the option to be written to standard output; for example:

```
lpstat –u
```

writes the status of all output requests that are still in the output queue.

–a[*list* **]** Write the acceptance status of destinations for output requests. The *list* argument is a list of intermixed printer names and class names.

–c[*list* **]** Write the class names and their members. The *list* argument is a list of class names.

–d Write the system default destination for output requests.

–o[*list* **]** Write the status of output requests. The *list* argument is a list of intermixed printer names, class names and request *ID*s.

–p[*list* **]** Write the status of printers. The *list* argument is a list of printer names.

–r Write the status of the line printer request scheduler.

–s Write 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 Write all status information.

–u[*list* **]** Write the status of output requests for users. The *list* argument is a list of login names.

–v[*list* **]** Write the names of printers and the pathnames of the devices associated with them. The *list* argument is a list of printer names.

OPERANDS

The following operand is supported:

ID A request *ID*, as returned by *lp*.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *lpstat*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME

Determine the format of date and time strings output when displaying line printer status information with the **−a**, **−o**, **−p**, **−t** or **−u** options.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file containing the information described in **OPTIONS**, in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *lpstat* utility cannot reliably determine the status of print requests in all conceivable circumstances. When the printer is under the control of another operating system or resides on a

remote system across a network, it need not be possible to determine the status of the print job after it has left the control of the local operating system. Even on local printers, spooling hardware in the printer may make it appear that the print job has been completed long before the final page is printed.

EXAMPLES

1. Obtain the status of two printers, the pathnames of two printers, a list of all class names and the status of the request named **HiPri-33**:

```
lpstat -plaser1,laser4 -v"laser2 laser3" -c HiPri-33
```

OB 2. Obtain user print job status using the obsolescent mixed blank and comma form:

```
lpstat –u"ddg,gmv, maw"
```

FUTURE DIRECTIONS

A version of *lpstat* that fully supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** may be introduced in a future issue.

SEE ALSO

cancel, *lp*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

NAME

ls – list directory contents

SYNOPSIS

EX `ls [-CFRacdilqrtu1][-fgmnopsx][file...]`

DESCRIPTION

For each operand that names a file of a type other than directory, *ls* writes the name of the file as well as any requested, associated information. For each operand that names a file of type directory, *ls* writes the names of files contained within that directory, as well as any requested, associated information.

If no operands are specified, the contents of the current directory are written. If more than one operand is specified, non-directory operands are written first; directory and non-directory operands are sorted separately according to the collating sequence in the current locale.

OPTIONS

The *ls* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–C Write multi-text-column output with entries sorted down the columns, according to the collating sequence. The number of text columns and the column separator characters are unspecified, but should be adapted to the nature of the output device.

–F Write a slash (/) immediately after each pathname that is a directory, an asterisk (*)
EX after each that is executable, and a vertical bar (|) after each that is a FIFO. For other file types, other symbols may be written.

–R Recursively list subdirectories encountered.

–a Write out all directory entries, including those whose names begin with a period (.). Entries beginning with a period will not be written out unless explicitly referenced, the **–a** option is supplied, or an implementation-dependent condition causes them to be written.

–c Use time of last modification of the file status information (see **<sys/stat.h>** in the **XSH** specification) instead of last modification of the file itself for sorting (**–t**) or writing (**–l**).

–d Do not treat directories differently from other types of files. The use of **–d** with **–R** produces unspecified results.

EX **–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.

EX **–g** The same as **–l**, except that the owner is not written.

–i For each file, write the file's file serial number (see *stat*( ) in the **XSH** specification).

–l (The letter ell.) Write out in long format (see **STDOUT**). When **–l** (ell) is specified, **–1** (one) is assumed.

EX **–m** Stream output format; list files across the page, separated by commas.

EX **–n** The same as **–l**, except that the owner's UID and GID numbers are written, rather than the associated character strings.

EX **–o** The same as **–l**, except that the group is not written.

EX **–p** Write a slash (/) after each filename if that file is a directory.

–q Force each instance of non-printable filename characters and tab characters to be written as the question-mark (?) character. Implementations may provide this option by default if the output is to a terminal device.

–r Reverse the order of the sort to get reverse collating sequence or oldest first.

EX **–s** Indicate the total number of file system blocks consumed by each file displayed. The block size is implementation-dependent.

–t Sort by time modified (most recently modified first) before sorting the operands by the collating sequence.

–u Use time of last access (see **<sys/stat.h>** in the **XSH** specification) instead of last modification of the file for sorting (**–t**) or writing (**–l**).

EX **–x** The same as **–C**, except that the multi-text-column output is produced with entries sorted across, rather than down, the columns,

–1 (The numeric digit one.) Force output to be one entry per line.

EX Specifying more than one of the options in the following mutually exclusive pairs is not considered an error: **–C** and **–l** (ell), **–m** and **–l** (ell), **–x** and **–l** (ell), **–C** and **–1** (one), **–c** and **–u**. The last option specified in each pair determines the output format.

OPERANDS

The following operand is supported:

file A pathname of a file to be written. If the file specified is not found, a diagnostic message will be output on standard error.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ls*:

COLUMNS

Determine the user's preferred column position width for writing multiple text-column output. If this variable contains a string representing a decimal integer, the *ls* utility calculates how many pathname text columns to write (see **–C**) based on the width provided. If *COLUMNS* is not set or invalid, an implementation-dependent number of column positions is assumed, based on the implementation's knowledge of the output device. The column width chosen to write the names of files in any given directory will be constant. Filenames will not be truncated to fit into the multiple text-column output.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for character collation information in determining the pathname collation sequence.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments) and which characters are defined as printable (character class **print**).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format and contents for date and time strings written by *ls*.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone for date and time strings written by *ls*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The default format is to list one entry per line to standard output; the exceptions are to terminals
EX or when one of the **–C**, **–m** or **–x** options is specified. If the output is to a terminal, the format is implementation-dependent.

EX When **–m** is specified, the format used is:

```
"%s, %s, ...\n", <filename1>, <filename2>
```

where the largest number of filenames is written without exceeding the length of the line.

If the **–i** option is specified, the file's file serial number (see **<sys/stat.h>** in the **XSH** specification) is written in the following format before any other output for the corresponding entry:

```
"%u ", <file serial number>
```

If the **–l** option is specified, the following information will be written:

```
"%s %u %s %s %u %s %s\n", <file mode>, <number of links>,
<owner name>, <group name>, <number of bytes in the file>,
<date and time>, <pathname>
```

EX The **–g**, **–n** and **–o** options use the same format as **–l**, but with omitted items and their associated blank characters; see **OPTIONS**.

EX If <*owner name*> or <*group name*> cannot be determined, or if **–n** is given, they are replaced with their associated numeric values using the format "%u".

The <*date and time*>, field will contain the appropriate date and timestamp of when the file was last modified. In the POSIX locale, the field is the equivalent of the output of the following *date* command:

```
date "+%b %e %H:%M"
```

if the file has been modified in the last six months, or:

```
date "+%b %e  %Y"
```

(where two space characters are used between **%e** and **%Y)** if the file has not been modified in

the last six months or if the modification date is in the future, except that, in both cases, the final newline character produced by *date* is not included and the output is as if the *date* command were executed at the time of the last modification date of the file rather than the current time. When the LC_TIME locale category is not set to the POSIX locale, a different format and order of presentation of this field may be used.

If the file is a character special or block special file, the size of the file may be replaced with implementation-dependent information associated with the device in question.

If the pathname was specified as a *file* operand, it will be written as specified.

EX The file mode written under the **–l**, **–g**, **–n** and **–o** options consists of the following format:

```
"%c%s%s%s%c", <entry type>, <owner permissions>,
<group permissions>, <other permissions>,
<optional alternate access method flag>
```

The *<optional alternate access method flag>* is a single space character if there is no alternate or additional access control method associated with the file; otherwise, a printable character is used.

The *<entry type>* character describes the type of file, as follows:

d directory
b block special file
c character special file
p FIFO
– regular file

Implementations may add other characters to this list to represent other, implementation-dependent, file types.

The next three fields are three characters each:

<owner permissions>
Permissions for the file owner class (see **file access permissions** in the **XBD** specification, **Chapter 2**, **Glossary**).

<group permissions>
Permissions for the file group class.

<other permissions>
Permissions for the file other class.

Each field has three character positions:

1. If r, the file is readable; if –, it is not readable.
2. If w, the file is writable; if –, it is not writable.
3. The first of the following that applies:

   S If in *<owner permissions>*, the file is not executable and set-user-ID mode is set. If in *<group permissions>*, the file is not executable and set-group-ID mode is set.

   s If in *<owner permissions>*, the file is executable and set-user-ID mode is set. If in *<group permissions>*, the file is executable and set-group-ID mode is set.

   x The file is executable or the directory is searchable.

– None of the attributes of S, s or x applies.

Implementations may add other characters to this list for the third character position. Such additions will, however, be written in lower-case if the file is executable or searchable, and in upper-case if it is not.

EX If any of the **–l**, **–g**, **–n**, **–o** or **–s** options is specified, each list of files within the directory will be preceded by a status line indicating the number of file system blocks occupied by files in the directory in 512-byte units, rounded up to the next integral number of units, if necessary. In the POSIX locale, the format is:

```
"total %u\n", <number of units in the directory>
```

If more than one directory, or a combination of non-directory files and directories are written, either as a result of specifying multiple operands, or the **–R** option, each list of files within a directory will be preceded by:

```
"\n%s:\n", <directory name>
```

If this string is the first thing to be written, the first newline character is not written. This output precedes the number of units in the directory.

EX If the **–s** option is given, each file shall be written with the number of blocks used by the file. Along with **–C**, **–1**, **–m** or **–x**, the number and a space character precede the filename; with **–g**, **–l**, **–n** or **–o**, they precede each line describing a file.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files were written successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Many implementations use the equal sign (=) and the at sign (@) to denote sockets bound to the file system and symbolic links, respectively, for the **–F** option. Similarly, many historical implementations use the s character and the l character to denote sockets and symbolic links, respectively, as the entry type characters for the **–l** option.

It is difficult for an application to use every part of the file modes field of *ls* **–l** in a portable manner. Certain file types and executable bits are not guaranteed to be exactly as shown, as implementations may have extensions. Applications can use this field to pass directly to a user printout or prompt, but actions based on its contents should generally be deferred, instead, to the *test* utility.

The output of *ls* (with the **–l** and related options) contains information that logically could be used by utilities such as *chmod* and *touch* to restore files to a known state. However, this information is presented in a format that cannot be used directly by those utilities or be easily translated into a format that can be used. A character has been added to the end of the

permissions string so that applications will at least have an indication that they may be working in an area they do not understand instead of assuming that they can translate the permissions string into something that can be used. Future issues or related documents may define one or more specific characters to be used based on different standard additional or alternative access control mechanisms.

As with many of the utilities that deal with filenames, the output of *ls* for multiple files or in one of the long listing formats must be used carefully on systems where filenames can contain embedded white space. Systems and system administrators should institute policies and user training to limit the use of such filenames.

The number of disk blocks occupied by the file that it reports varies depending on underlying file system type, block size units reported and the method of calculating the number of blocks. On some file system types, the number is the actual number of blocks occupied by the file (counting indirect blocks and ignoring holes in the file); on others it is calculated based on the file size (usually making an allowance for indirect blocks, but ignoring holes). The former is probably more useful, but depends on information not required by the **XSH** specification and not readily accessible on some file system types. Therefore, applications cannot depend on **–s** to provide any portable information. Applications should use the file size reported by the **–l** option in any calculations about the space needed to store a file.

EXAMPLES

An example of a small directory tree being fully listed with *ls* **–laRF a** in the POSIX locale:

```
total 11
drwxr-xr-x   3 hlj      prog          64 Jul  4 12:07 ./
drwxrwxrwx   4 hlj      prog        3264 Jul  4 12:09 ../
drwxr-xr-x   2 hlj      prog          48 Jul  4 12:07 b/
-rwxr--r--   1 hlj      prog         572 Jul  4 12:07 foo*

a/b:
total 4
drwxr-xr-x   2 hlj      prog          48 Jul  4 12:07 ./
drwxr-xr-x   3 hlj      prog          64 Jul  4 12:07 ../
-rw-r--r--   1 hlj      prog         700 Jul  4 12:07 bar
```

FUTURE DIRECTIONS

The **–s** uses implementation-dependent units and cannot be used portably; it will be withdrawn from a future issue.

SEE ALSO

chmod, *du*, *find*, the **XSH** specification description of **<sys/stat.h>**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

m4 – macro processor (**DEVELOPMENT**)

SYNOPSIS

EX `m4 [-s][ -D name[=val]]...[-U name]... file...`

DESCRIPTION

The *m4* utility is a macro processor that reads one or more text files, processes them according to their included macro statements, and writes the results to standard output.

OPTIONS

The *m4* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the order of the –**D** and –**U** options is significant. The following options are supported:

–**s** Enable line synchronisation output for the *c89* preprocessor phase (that is, **#line** directives).

–**D** *name*[=*val*]
Define *name* to *val* or to null if =*val* is omitted.

–**U** *name*
Undefine *name*.

OPERANDS

The following operand is supported:

file A pathname of a text file to be processed. If no *file* is given, or if it is –, the standard input is read.

STDIN

The standard input is a text file that is used if no *file* operand is given, or if it is –.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *m4*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is the same as the input files, after being processed for macro expansion.

STDERR

Used to display strings with the **errprint** macro, macro tracing enabled by the **traceon** macro, or for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *m4* utility compares each token from the input against the set of built-in and user-defined macros. If the token matches the name of a macro, then the token is replaced by the macros defining text, if any, and rescanned for matching macro names. Once no portion of the token matches the name of a macro, it is written to standard output. Macros may have arguments, in which case the arguments will be substituted into the defining text before it is rescanned.

Macro calls have the form:

name(*arg1, arg2, . . ., argn*)

Macro names consist of letters, digits and underscores, where the first character is not a digit. Tokens not of this form are not treated as macro names.

The left parenthesis must immediately follow the name of the macro. If a token matching the name of a macro is not followed by a left parenthesis, it will be handled as a use of that macro without arguments.

If a macro name is followed by a left parenthesis, its arguments are the comma-separated tokens between the left parenthesis and the matching right parenthesis. Unquoted blank and newline characters preceding each argument are ignored. All other characters, including trailing blank and newline characters, are retained. Commas enclosed between left and right parenthesis characters do not delimit arguments.

Arguments are positionally defined and referenced. The string $1 in the defining text will be replaced by the first argument. Systems support at least nine arguments; only the first nine can be referenced, using the strings $1 to $9, inclusive. The string $0 will be replaced with the name of the macro. The string $# will be replaced by the number of arguments as a string. The string $* will be replaced by a list of all of the arguments, separated by commas. The string $@ will be replaced by a list of all of the arguments separated by commas, and each argument will be quoted using the current left and right quoting strings.

If fewer arguments are supplied than are in the macro definition, the omitted arguments are taken to be null. It is not an error if more arguments are supplied than are in the macro definition.

No special meaning is given to any characters enclosed between matching left and right quoting strings, but the quoting strings are themselves discarded. By default, the left quoting string consists of a grave accent (‘) and the right quoting string consists of an acute accent (’) see also the **changequote** macro.

Comments are written but not scanned for matching macro names; by default, the begin-comment string consists of the number sign character and the end-comment string consists of a newline character. See also the **changecom** and **dnl** macros.

The *m4* utility makes available the following built-in macros. They can be redefined, but once this is done the original meaning is lost. Their values are null unless otherwise stated.

changecom
The **changecom** macro sets the begin- and end-comment strings. With no arguments, the comment mechanism is disabled. With a single argument, that argument becomes the begin-comment string and the newline character becomes the end-comment string. With two arguments, the first argument becomes the begin-comment string and the second argument becomes the end-comment string. Systems support comment strings of at least five characters.

changequote
The **changequote** macro sets the begin- and end-quote strings. With no arguments, the quote strings are set to the default values (that is, `´). With a single argument, that argument becomes the begin-quote string and the newline character becomes the end-quote string. With two arguments, the first argument becomes the begin-quote string and the second argument becomes the end-quote string. Systems support quote strings of at least five characters.

decr The defining text of the **decr** macro is its first argument decremented by 1. It is an error to specify an argument containing any non-numeric characters.

define The second argument is specified as the defining text of the macro whose name is the first argument.

defn The defining text of the **defn** macro is the quoted definition (using the current quoting strings) of its arguments.

divert The *m4* utility maintains ten temporary buffers, numbered 0 to 9, inclusive. When the last of the input has been processed, any output that has been placed in these buffers will be written to standard output in buffer-numerical order. The **divert** macro diverts future output to the buffer specified by its argument. Specifying no argument or an argument of 0 resumes the normal output process. Output diverted to a stream other than 0 to 9 is discarded. It is an error to specify an argument containing any non-numeric characters.

divnum The defining text of the **divnum** macro is the number of the current output stream as a string.

dnl The **dnl** macro causes *m4* to discard all input characters up to and including the next newline character.

dumpdef
The **dumpdef** macro writes the defined text for each of the macros specified as arguments, or, if no arguments are specified, for all macros.

errprint The **errprint** macro writes its arguments to standard error.

eval The **eval** macro evaluates its first argument as an arithmetic expression, using 32-bit signed integer arithmetic. All of the C-language operators are supported, except for [ ], –>, ++, – –, (*type*), unary *, &, **sizeof**, ‘‘ , ’’, **?:** and all assignment operators. It is an error to specify any of these operators. Precedence and associativity are as in C. Systems support octal and hexadecimal numbers as in C. The second argument, if specified, sets the radix for the result; the default is 10. The third argument, if specified, sets the minimum number of digits in the result. It is an error to specify an argument containing any non-numeric characters.

ifdef If the first argument to the **ifdef** macro is defined and it is not defined to be zero, the defining text is the second argument. Otherwise, the defining text is the third argument, if specified, or the null string, if not.

ifelse If the first argument (or the defining text of the first argument if it is a macro name) to the **ifelse** macro is the same as the second argument (or the defining text of the second argument if it is a macro name), then the defining text is the third argument. If there are more than four arguments, the initial comparison of the first and second arguments are repeated for each group of three arguments. If no match is found, the defining text will be the argument following the last set of three compared, otherwise it will be null.

include The defining text for the **include** macro is the contents of the file named by the first argument. It is an error if the file cannot be read.

incr The defining text of the **incr** macro is its first argument incremented by 1. It is an error to specify an argument containing any non-numeric characters.

index The defining text of the **index** macro is the first character position (as a string) in the first argument where a string matching the second argument begins (zero origin), or –1 if the second argument does not occur.

len The defining text of the **len** macro is the length (as a string) of the first argument.

m4exit Exit from the *m4* utility. If the first argument is specified, it will be the exit code. The default is zero. It is an error to specify an argument containing any non-numeric characters.

m4wrap The first argument will be processed when EOF is reached. If the **m4wrap** macro is used multiple times, the arguments specified will be processed in the order in which the **m4wrap** macros were processed.

maketemp
The defining text is the first argument, with any trailing capital X characters replaced with the current process ID as a string.

popdef The **popdef** macro deletes the current definition of its arguments, replacing it with the previous one. If there is no previous definition, the macro is undefined.

pushdef
The **pushdef** macro is identical to the **define** macro with the exception that it preserves any current definition for future retrieval using the **popdef** macro.

shift The defining text for the **shift** macro is all of its arguments except for the first one.

sinclude
The **sinclude** macro is identical to the **include** macro, except that it is not an error if the file is inaccessible.

substr The defining text for the **substr** macro is the substring of the first argument beginning at the zero-offset character position specified by the second argument. The third argument, if specified, is the number of characters to select; if not specified, the characters from the starting point to the end of the first argument become the defining text. It is not an error to specify a starting point beyond the end of the first argument and the defining text will be null. It is an error to specify an argument containing any non-numeric characters.

syscmd The **syscmd** macro interprets its first argument as a shell command line. The defining text is the string result of that command. No output redirection is performed by the *m4* utility. The exit status value from the command can be retrieved using the **sysval**

macro.

sysval The defining text of the **sysval** macro is the exit value of the utility last invoked by the **syscmd** macro (as a string).

traceon The **traceon** macro enables tracing for the macros specified as arguments, or, if no arguments are specified, for all macros. The trace output is written to standard error in an unspecified format.

traceoff The **traceoff** macro disables tracing for the macros specified as arguments, or, if no arguments are specified, for all macros.

translit The defining text of the **translit** macro is the first argument with every character that occurs in the second argument replaced with the corresponding character from the third argument.

undefine
The **undefine** macro deletes all definitions (including those preserved using the **pushdef** macro) of the macros named by its arguments.

undivert
The **undivert** macro causes immediate output of any text in temporary buffers named as arguments, or all temporary buffers if no arguments are specified. Buffers can be undiverted into other temporary buffers. Undiverting discards the contents of the temporary buffer. It is an error to specify an argument containing any non-numeric characters.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred

If the m4exit macro is used, the exit value can be specified by the input file.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **defn** macro is useful for renaming macros, especially built-ins.

EXAMPLES

An example of a single *m4* input file capable of generating two output files follows. The file **file1.m4** could contain lines such as:

```
if(VER, 1, do_something)
if(VER, 2, do_something)
```

The makefile for the program might include:

```
file1.1.c :  file1.m4
             m4 -D VER=1 file1.m4 > file1.1.c
             ...
file1.2.c :  file1.m4
             m4 -D VER=2 file1.m4 > file1.2.c
             ...
```

The –**U** option can be used to undefine **VER**. If **file1.m4** contains:

```
if(VER, 1, do_something)
if(VER, 2, do_something)
ifndef(VER, do_something)
```

then the makefile would contain:

```
file1.0.c :   file1.m4
              m4 -U VER file1.m4 > file1.0.c
              ...
file1.1.c :   file1.m4
              m4 -D VER=1 file1.m4 > file1.1.c
              ...
file1.2.c :   file1.m4
              m4 -D VER=2 file1.m4 > file1.2.c
              ...
```

FUTURE DIRECTIONS

None.

SEE ALSO

c89.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guideline support mandated.

Internationalised environment variable support mandated.

NAME

mail – send or read mail (**TO BE WITHDRAWN**)

SYNOPSIS

EX `mail [-e] [-f file]`

UN EX `mail [-e| -p][-qr][-f file]`

UN EX `mail [-t] name ...`

DESCRIPTION

Reading Mail

The *mail* utility without arguments writes a user's mail to standard output, message-by-message. Mail is stored in the user's individual mailfile. For each message, the user is given a prompt and a line is read from the standard input to determine the disposition of the message; see **EXTENDED DESCRIPTION**.

Sending Mail

When *name*s (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 sequence **From** are preceded with a >.

If a user being sent mail is not recognised, or if *mail* is interrupted during input, the message is saved in the file **dead.letter** to allow editing and resending. Note that this is regarded as a temporary file in that it is recreated every time it is needed, erasing the previous contents of **dead.letter**. The *mail* utility tries to create **dead.letter** in the current directory. If that fails, it tries to create **dead.letter** in the directory specified by the *HOME* environment variable.

It may also be possible to send mail to remote systems using system-specific naming conventions.

There are implementation-specific mechanisms that can be used to cause all mail sent to the user to be forwarded to one or more other destinations.

OPTIONS

The *mail* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported for reading mail:

–e Test for the presence of mail. Display nothing and exit with a successful return code if there is mail to read.

UN **–p** Write all mail to standard output without prompting for disposition.

UN **–q** Terminate after interrupts. By default, an interrupt terminates only the message being written.

UN **–r** Write messages in first-in, first-out order.

–f *file* Use *file* (for example, **mbox**) instead of the default mailfile.

The following option is supported for sending mail:

UN **–t** Precede the message by a list of all users the *mail* is sent to.

OPERANDS

The following operand is supported for sending mail:

name A user login name.

STDIN

The standard input is a text file of commands for writing mail or a text file to be added to the user's mailfile when sending mail.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variable affects the execution of *mail*:

HOME Determine the pathname of the user's home directory.

The following environment variables may affect the execution of *mail*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME

Determine the format and contents of date and time strings displayed by the *mail* utility.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS

When reading mail, signals are caught and the user is returned immediately to the next prompt; however, if the **–q** option is specified, *mail* terminates. When sending mail, signals are caught, interrupting the user's input. Any text already input is saved in the file **dead.letter**.

STDOUT

When reading mail, the standard output is used for prompting and writing mail messages; the format of prompting and other informational messages is unspecified. When sending mail, the standard output is not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

When reading mail, messages can be appended to files designated by the **s** or **w** commands or to a user's mailfile by using the **m** command. When sending mail, messages are written to mailfiles or to **dead.letter**.

EXTENDED DESCRIPTION

When reading mail, the following commands read from the standard input determine the disposition of messages:

newline
: Go on to next message.

+ : Same as the newline character.

d : Delete message and go on to next message.

p : Display message again.

– : Go back to previous message.

s[*file* **]** : Save the message in the named *file* (**mbox** is default).

PI **w[** *file* **]** : Save the message (on some implementations, without its header) in the named *file* (**mbox** is default).

PI **m[** *name* . . . **]**
: Mail the message to the named users; the default is the user who invoked *mail*.

PI **q** : Store undeleted mail and stop.

<EOF> : Same as **q**.

x : Put all mail back in the mailfile unchanged and stop.

!*command*
: Escape to the command interpreter to execute *command*.

UN ***** : Display a command summary.

EXIT STATUS

The following exit values are returned:

0 Successful completion when the user had mail.
1 The user had no mail or an initialisation error occurred.
>1 An error occurred after initialisation.

CONSEQUENCES OF ERRORS

When reading mail, the mailfile is unchanged. When sending mail, some of the named users need not have their mailfiles appended with the message.

APPLICATION USAGE

Delivery of messages to remote systems requires the existence of communication paths to such systems. These need not exist.

The location of stored mail on exiting from *mail* using the **q** command differs between implementations and may be either the user's mailfile or the user's **mbox**.

In the description of reading mail, the phrase ''go on to next message'' may or may not imply the displaying of the next message.

Input lines are limited to {LINE_MAX} bytes, but mailers between systems may impose more severe line-length restrictions.

The *mail* utility cannot guarantee support for all character encodings in all circumstances. For example, inter-system mail may be restricted to 7-bit data by the underlying network, 8-bit data need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used.

EXAMPLES

None.

FUTURE DIRECTIONS

This utility is being withdrawn from a future issue. The *mailx* utility should be used instead.

SEE ALSO

mailx, *uuencode*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Marked **TO BE WITHDRAWN**.

NAME

mailx – process messages

SYNOPSIS

Send Mode:

```
mailx [-s subject] address...
```

Receive Mode:

```
mailx -e
```

EX

```
mailx [-HiNn][-F][-u user]
```

EX

```
mailx -f[-HiNn][-F][file]
```

DESCRIPTION

The *mailx* utility provides a message sending and receiving facility. It has two major modes, selected by the options used: Send Mode and Receive Mode.

Send Mode

Send Mode can be used by applications or users to send messages from the text in standard input.

Receive Mode

Receive Mode is more oriented to interactive users. Mail can be read and sent in this interactive mode.

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 one or more unspecified locations for each user, collectively called the system *mailbox* for that user. When *mailx* is invoked in Receive Mode, the system mailbox is the default place to find them. As messages are read, they will be marked to be moved to a secondary file for storage, unless specific action is taken. This secondary file is called the **mbox** and is normally located in the *HOME* directory of the user (see *MBOX* in **ENVIRONMENT VARIABLES** for a description of this file). Messages remain in this file until explicitly removed. When the **–f** option is used to read mail messages from secondary files, messages will be retained in those files unless specifically removed. All three of these locations system mailbox, **mbox** and secondary file are referred to in this section as simply ‘‘mailboxes,’’ unless more specific identification is required.

OPTIONS

The *mailx* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–e Test for the presence of mail in the system mailbox. The *mailx* utility will write nothing and exit with a successful return code if there is mail to read.

–f Read messages from the file named by the *file* operand instead of the system mailbox. (See also **folder**.) If no *file* operand is specified, read messages from the **mbox** instead of the system mailbox.

EX **–F** Record the message in a file named after the first recipient. The name is the login-name portion of the address found first on the To: line in the mail header. Overrides the **record** variable, if set (see **Internal Variables in mailx** on page 465.)

–H Write a header summary only.

–i Ignore interrupts. (See also **ignore**).

–n Do not initialise from the system default start-up file. See **EXTENDED DESCRIPTION**.

–N Do not write an initial header summary.

–s *subject*
Set the Subject header field to *subject*. All characters in the *subject* string will appear in the delivered message. The results are unspecified if *subject* is longer than {LINE_MAX} – 10 bytes or contains a newline character.

–u *user*
Read the system mailbox of the login name *user*. This will only be successful if the invoking user has the appropriate privileges to read the system mailbox of that user.

OPERANDS

The following operands are supported:

address Addressee of message. When **–n** is specified and no user start-up files are accessed (see **EXTENDED DESCRIPTION**), this must be an address to pass to the mail delivery system. Any system or user start-up files may enable aliases (see **alias** under **Commands in mailx** on page 468) that may modify the form of *address* before it is passed to the mail delivery system.

file A pathname of a file to be read instead of the system mailbox when **–f** is specified. The meaning of the *file* option-argument is affected by the contents of the **folder** internal variable; see **Internal Variables in mailx** on page 465.

STDIN

When *mailx* is invoked in Send Mode (the first synopsis line), standard input must be the message to be delivered to the specified addresses. In both Send and Receive Modes, standard-input lines beginning with the escape character (usually tilde (~)) affect processing as described in **Command Escapes in mailx** on page 475.

INPUT FILES

When *mailx* is used as described by this document, the *file* option-argument (see the **–f** option) and the **mbox** must be text files containing mail messages, formatted as described in **OUTPUT FILES**. The nature of the system mailbox is unspecified; it need not be a file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *mailx*:

DEAD Determine the pathname of the file in which to save partial messages in case of interrupts or delivery errors. The default is **dead.letter** in the directory named by the *HOME* variable.

EDITOR
Determine the name of a utility to invoke when the **edit** (see **Commands in mailx** on page 468) or ~e (see **Command Escapes in mailx** on page 475) command is used. The default is unspecified.

HOME Determine the pathname of the user's home directory.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the handling of case-insensitive address and header-field comparisons.

LC_TIME
Determine the format and contents of the date and time strings written by *mailx*.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

LISTER Determine a string representing the command for writing the contents of the **folder** directory to standard output when the **folders** command is given (see **folders** in **Commands in mailx** on page 468). Any string acceptable as a *command_string* operand to the *sh* **–c** command is valid. If this variable is null or not set, the output command will be *ls*. The default value is unset.

MAILRC
Determine the pathname of the start-up file. The default is **.mailrc** in the *HOME* directory.

MBOX Determine a pathname of the file to save messages from the system mailbox that have been read. The **exit** command overrides this function, as will saving the message explicitly in another file. The default is **mbox** in the directory named by the *HOME* variable.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PAGER Determine a string representing an output filtering or pagination command for writing the output to the terminal. Any string acceptable as a *command_string* operand to the *sh* **–c** command is valid. When standard output is a terminal device, the message output will be piped through the command if the *mailx* internal variable **crt** is set to a value less the number of lines in the message; see **Internal Variables in mailx** on page 465. If the *PAGER* variable is null or not set, the paginator will be either *more* or another paginator utility documented in the system documentation.

SHELL Determine the name of a preferred command interpreter. The default is *sh*.

TERM Determine the name of the terminal type, to indicate in an unspecified manner, if the internal variable **screen** is not specified, the number of lines in a screenful of headers. If *TERM* is not set or is set to null, an unspecified default terminal type will be used and the value of a screenful is unspecified.

VISUAL

Determine a pathname of a utility to invoke when the **visual** command (see **Commands in mailx** on page 468) or ~v command-escape (see **Command Escapes in mailx** on page 475) is used. If this variable is null or not set, the full-screen editor will be *vi*.

ASYNCHRONOUS EVENTS

When *mailx* is in Send Mode and standard input is not a terminal, it takes the standard action for all signals.

In Receive Mode, or in Send Mode when standard input is a terminal, if a SIGINT signal is received:

1. If in command mode, the current command, if there is one, will be aborted, and a command-mode prompt will be written.
2. If in input mode:
   a. If **ignore** is set, *mailx* will write @\n, discard the current input line, and continue processing, bypassing the message-abort mechanism described in item 2b.
   b. If the interrupt was received while sending mail, either when in Receive Mode or in Send Mode, a message will be written, and another subsequent interrupt, with no other intervening characters typed, will be required to abort the mail message. If in Receive Mode and another interrupt is received, a command-mode prompt will be written. If in Send Mode and another interrupt is received, *mailx* will terminate with a non-zero status.

      In both cases listed in item b, if the message is not empty:

      i. If **save** is enabled and the file named by *DEAD* can be created, the message will be written to the file named by *DEAD*. If the file exists, the message will be written to replace the contents of the file.

      ii. If **save** is not enabled, or the file named by *DEAD* cannot be created, the message will not be saved.

The *mailx* utility takes the standard action for all other signals.

STDOUT

In command and input modes, all output, including prompts and messages, is written to standard output.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Various *mailx* commands and command escapes can create or add to files, including the **mbox**, the dead-letter file and secondary mailboxes. When *mailx* is used as described in this document, these files will be text files, formatted as follows:

line beginning with From<space>
[one or more *header-lines*; see **Commands in mailx** on page 468**]**
empty line
[zero or more *body lines*
*empty line***]**
[*line beginning with* From<space>...**]**

where each message begins with the **From**<space> line shown, preceded by the beginning of the file or an empty line. (The **From**<space> line is considered to be part of the message header, but

not one of the header-lines referred to in **Commands in mailx** on page 468; thus, it is not affected by the **discard**, **ignore** or **retain** commands.) The formats of the remainder of the **From**<space> line and any additional header lines are unspecified, except that none will be empty. The format of a message body line is also unspecified, except that no line following an empty line can start with **From**<space>; *mailx* will modify any such user-entered message body lines (following an empty line and beginning with **From**<space>) by adding one or more characters to precede the F; it may add these characters to **From**<space> lines that are not preceded by an empty line.

When a message from the system mailbox or entered by the user is not a text file, it is implementation-dependent how such a message is stored in files written by *mailx*.

EXTENDED DESCRIPTION

When *mailx* is invoked using one of the Receive Mode synopsis forms, it will write the page of header-summary lines (see below) containing the first new message (if **–N** is not specified), or the first unread message if there are no new messages, or the first message if there are no new or unread messages, followed by a prompt indicating *mailx* can accept regular commands (see **Commands in mailx** on page 468); this is termed *command mode*. When *mailx* is invoked using the Send Mode synopsis and standard input is a terminal, if no subject is specified on the command line and the **asksub** variable is set, a prompt for the subject will be written. At this point *mailx* is in *input mode*. This input mode is also entered when using one of the Receive
EX Mode synopsis forms and a reply or new message is composed using the **reply**, **Reply**, **followup**, **Followup** or **mail** commands. When the message is typed and the end of message is encountered, the message will be passed to the mail delivery software. Commands can be entered by beginning a line with the escape character (by default, tilde (~)) followed by a single command letter and optional arguments. See **Command Escapes in mailx** on page 475 for a summary of these commands.

Note: For notational convenience, this section uses the default escape character, tilde, in all references and examples.

At any time, the behaviour of *mailx* is governed by a set of environmental and internal variables. These are flags and valued parameters that can be set and cleared via the *mailx* **set** and **unset** commands.

Regular commands are of the form:

[*command***] [***msglist***] [***argument ...***]**

If no *command* is specified in command mode, **print** is assumed. In input mode, commands are recognised by the escape character, and lines not treated as commands are taken as input for the message.

In command mode, each message will be assigned a sequential number, starting with 1.

All messages have a state that affects how they are displayed in the header summary and how they are retained or deleted upon termination of *mailx*. There is at any time the notion of a *current* message, marked by a > at the beginning of a line in the header summary. All messages are in one of the following states:

new The message is present in the system mailbox and has not been viewed by the user or moved to any other state. Messages in state *new* when *mailx* quits will be retained in the system mailbox.

unread The message has been present in the system mailbox for more than one invocation of *mailx* and has not been viewed by the user or moved to any other state. Messages in state *unread* when *mailx* quits will be retained in the system mailbox.

read The message has been processed by one of the following commands: **~f**, **~m**, **~F**, **~M**, **copy**, **mbox**, **next**, **pipe**, **print**, **Print**, **top**, **type**, **Type**, **undelete**. The **delete**, **dp** and **dt** commands may also cause the next message to be marked as *read*, depending on the value of the **autoprint** variable. Messages that are in the system mailbox and in state *read* when *mailx* quits will be saved in the **mbox**, unless the internal variable **hold** was set. Messages that are in the **mbox** or in a secondary mailbox and in state *read* when *mailx* quits will be retained in their current location.

deleted The message has been processed by one of the following commands: **delete**, **dp**, **dt**. A message processed by **save** will be in state *deleted* unless the internal variable **keepsave** was set. Messages in state *deleted* when *mailx* quits will be deleted.

preserved
The message has been processed by a **preserve** command. When *mailx* quits, the message will be retained in its current location.

The header-summary line for each message will indicate the state of the message.

Many commands take an optional list of messages (*msglist*) on which to operate, which defaults to the current message. A *msglist* is a list of message specifications separated by blank characters, which can include:

n Message number *n*.

+ The next undeleted message, or the next deleted message for the **undelete** command.

– The next previous undeleted message, or the next previous deleted message for the **undelete** command.

. The current message.

^ The first undeleted message, or the first deleted message for the **undelete** command.

$ The last message.

***** All messages.

n–m An inclusive range of message numbers.

address All messages from *address*; any address as shown in a header summary will be matchable in this form.

/string All messages with *string* in the subject line (case ignored).

:*c* All messages of type *c*, where *c* must be one of:

d deleted messages

n new messages

o old messages (any not in state *read* or *new*)

r read messages

u unread messages

Other commands take an optional message (*message*) on which to operate, which defaults to the current message. All of the forms allowed for *msglist* are also allowed for *message*, but if more than one message is specified, only the first will be operated on.

Other arguments are usually arbitrary strings whose usage depends on the command involved.

Start-up in mailx

At start-up time, *mailx* will take the following steps in sequence:

1. Establish all variables at their stated default values.
2. Process command-line options, overriding corresponding default values.
3. Import any of the *DEAD*, *EDITOR*, *MBOX*, *LISTER*, *PAGER*, *SHELL* or *VISUAL* variables that are present in the environment, overriding the corresponding default values.
4. Read *mailx* commands from an unspecified system start-up file, unless the **–n** option is given, to initialise any internal *mailx* variables and aliases.
5. Process the start-up file of *mailx* commands named in the user *MAILRC* variable.

Most regular *mailx* commands are valid inside start-up files, the most common use being to set up initial display options and alias lists. The following commands are invalid in the start-up file: EX **!**, **edit**, **hold**, **mail**, **preserve**, **reply**, **Reply**, **shell**, **visual**, **Copy**, **followup** and **Followup**. Any errors in the start-up file will either cause *mailx* to terminate with a diagnostic message and a non-zero status or to continue after writing a diagnostic message, ignoring the remainder of the lines in the start-up file.

A blank line in a start-up file is ignored.

Internal Variables in mailx

The following variables are internal *mailx* variables. Each internal variable can be set via the *mailx* **set** command at any time. The **unset** and **set no***name* commands can be used to erase variables.

In the following list, variables shown as:

```
variable
```

represent Boolean values. Variables shown as:

```
variable=value
```

will be assigned string or numeric values. For string values, the rules in **Commands in mailx** on page 468 concerning filenames and quoting also apply.

The defaults specified here may be changed by the implementation-specific system start-up file unless the user specifies the **–n** option.

EX **allnet** All network names whose login name components match are treated as identical. This causes the *msglist* message specifications to behave similarly. The default is **noallnet**. See also the **alternates** command and the **metoo** variable.

append Append messages to the end of the **mbox** file upon termination instead of placing them at the beginning. The default is **noappend**. This variable will not affect the **save** command when saving to the **mbox**.

ask

asksub Prompt for a subject line on outgoing mail if one is not specified on the command line with the **–s** option. The **ask** and **asksub** forms are synonyms; the system will refer to **asksub** and **noasksub** in its messages, but will accept **ask** and **noask** as user input to mean **asksub** and **noasksub**. It is not possible to set both **ask** and **noasksub**, or **noask** and **asksub**. The default is **asksub**, but no prompting will be done if standard input is not a terminal.

askbcc Prompt for the blind copy list. The default is **noaskbcc**.

askcc Prompt for the copy list. The default is **noaskcc**.

autoprint
Enable automatic writing of messages after **delete** and **undelete** commands. The default is **noautoprint**.

bang Enable the special-case treatment of exclamation-marks (!) in escape command lines; see the **escape** command and **Command Escapes in mailx** on page 475. The default is **nobang**, disabling the expansion of ! in the *command* argument to the ~! command and the ~<!*command* escape.

cmd=*command*
Set the default command to be invoked by the **pipe** command. The default is **nocmd**.

crt=*number*
Pipe messages having more than *number* lines through the command specified by the value of the *PAGER* variable. The default is **nocrt**. If it is set to null, the value used is implementation-dependent.

EX **debug** Enable verbose diagnostics for debugging. Messages are not delivered. The default is **nodebug**.

dot When **dot** is set, a period on a line by itself during message input from a terminal also signifies end-of-file (in addition to normal end-of-file). The default is **nodot**. If **ignoreeof** is set (see below), a setting of **nodot** will be ignored and the period is the only method to terminate input mode.

escape=*c*
Set the command escape character to be the character *c*. By default, the command escape character is tilde. If **escape** is unset, tilde will be used; if it is set to null, command escaping will be disabled.

flipr Reverse the meanings of the **R** and **r** commands. The default is **noflipr**.

folder=*directory*
The default directory for saving mail files. User-specified filenames beginning with a plus sign (+) will be expanded by preceding the filename with this directory name to obtain the real pathname. If *directory* does not start with a slash (/), the contents of *HOME* will be prefixed to it. The default is **nofolder**. If **folder** is unset or set to null, user-specified filenames beginning with + refer to files in the current directory that begin with the literal + character. See also **outfolder** below. The **folder** value need not affect the processing of the files named in *MBOX* and *DEAD*.

header Enable writing of the header summary when entering *mailx* in Receive Mode. The default is **header**.

hold Preserve all messages that are read in the system mailbox instead of putting them in the **mbox** save file. The default is **nohold**.

ignore Ignore interrupts while entering messages. The default is **noignore**.

ignoreeof
Ignore normal end-of-file during message input. Input can be terminated only by entering a period (.) on a line by itself or by the **~.** command escape. The default is **noignoreeof**. See also **dot** above.

indentprefix=*string*
A string that will be prefixed to each line that is inserted into the message by the **~m**

command escape. This variable defaults to one tab character.

keep When a system mailbox, secondary mailbox or **mbox** is empty, truncate it to zero length instead of removing it. The default is **nokeep**.

keepsave
Keep messages that have been saved in other files in the system mailbox instead of deleting them. The default is **nokeepsave**.

metoo Suppress the deletion of the login name of the user from the recipient list when replying to a message or sending to a group. The default is **nometoo**.

EX **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 (that is, one hop away). The default is **noonehop**.

outfolder
Cause the files used to record outgoing messages to be located in the directory specified by the **folder** variable unless the pathname is absolute. The default is **nooutfolder**. See the **record** variable.

page Insert a form-feed after each message sent through the pipe created by the **pipe** command. The default is **nopage**.

prompt=*string*
Set the command-mode prompt to *string*. If *string* is null or if **noprompt** is set, no prompting will occur. The default is to prompt with the string "? ".

quiet Refrain from writing the opening message and version when entering *mailx*. The default is **noquiet**.

record=*file*
Record all outgoing mail in the file with the pathname *file*. The default is **norecord**. See also **outfolder** above.

save Enable saving of messages in the dead-letter file on interrupt or delivery error. See the variable *DEAD* for the location of the dead-letter file. The default is **save**.

screen=*number*
Set the number of lines in a screenful of headers for the **headers** and z commands. If **screen** is not specified, a value based on the terminal type identified by the *TERM* environment variable, the window size, the baud rate, or some combination of these will be used.

EX **sendmail**=*shell-command*
Alternative command for delivering messages. The default is implementation-dependent **(TO BE WITHDRAWN)**.

EX **sendwait**
Wait for the background mailer to finish before returning. The default is **nosendwait**.

showto When the sender of the message was the user who is invoking *mailx*, write the information from the To: line instead of the From: line in the header summary. The default is **noshowto**.

sign=*string*
Set the variable inserted into the text of a message when the **~a** command escape is given. The default is **nosign**. The character sequences \t and \n are recognised in the

variable as tab and newline characters, respectively. (See also **~i** in **Command Escapes in mailx** on page 475.)

Sign=*string*
Set the variable inserted into the text of a message when the ~A command escape is given. The default is noSign. The character sequences \t and \n will be recognised in the variable as tab and newline characters, respectively.

toplines=*number*
Set the number of lines of the message to write with the **top** command. The default is 5.

Commands in mailx

The following *mailx* commands are provided. In the following list, header refers to lines from the message header, as shown in **OUTPUT FILES**. Header-line refers to lines within the header that begin with one or more non-white-space characters, immediately followed by a colon and white space and continuing until the next line beginning with a non-white-space character or an empty line. Header-field refers to the portion of a header line prior to the first colon in that line.

For each of the commands listed below, the command can be entered as the abbreviation (those characters in the Synopsis command word preceding the **[),** the full command (all characters shown for the command word, omitting the **[** and **]),** or any truncation of the full command down to the abbreviation. For example, the **exit** command (shown as **ex[it]** in the Synopsis) can be entered as **ex**, **exi** or **exit**.

The arguments to commands can be quoted, using the following methods:

- An argument can be enclosed between paired double-quotes (" ") or single-quotes (' '); any white space, shell word expansion or backslash characters within the quotes will be treated literally as part of the argument. A double-quote will be treated literally within single-quotes and *vice versa*. These special properties of the quote marks occur only when they are paired at the beginning and end of the argument.
- A backslash outside of the enclosing quotes is discarded and the following character treated literally as part of the argument.
- An unquoted backslash at the end of a command line is discarded and the next line continues the command.

Filenames, where expected, are subjected to the process of shell word expansions (see Section 2.6 on page 31); if more than a single pathname results and the command is expecting one file, the effects are unspecified. If the filename begins with an unquoted plus sign, it will not be expanded, but treated as the named file (less the leading plus) in the **folder** directory. (See the **folder** variable.)

Declare Aliases

Synopsis: `a[lias] [`*`alias`* `[`*`address...`*`]]`
Synopsis: `g[roup] [`*`alias`* `[`*`address...`*`]]`

Add the given addresses to the alias specified by *alias*. The names will be substituted when *alias* is used as a recipient address specified by the user in an outgoing message (that is, other recipients addressed indirectly through the **reply** command will not be substituted in this manner). Mail address alias substitution applies only when the alias string is used as a full address; for example, when **hlj** is an alias, hlj@posix.com does not trigger the alias substitution. If no arguments are given, write a listing of the current aliases to standard output. If only an *alias* argument is given, write a listing of the specified alias to standard output. These listings need not reflect the same order of addresses that were entered.

Declare Alternatives

Synopsis: `alt[ernates] name...`

Declare a list of alternative names for the user's login. When responding to a message, these names will be removed from the list of recipients for the response. The comparison of names will be in a case-insensitive manner. With no arguments, **alternates** will write the current list of alternative names.

Change Current Directory

Synopsis: `cd [directory]`
Synopsis: `ch[dir] [directory]`

Change directory. If *directory* is not specified, the contents of *HOME* will be used.

Copy Messages

Synopsis: `c[opy] [file]`
Synopsis: `c[opy] [msglist] file`
EX *Synopsis*: `C[opy] [msglist]`

Copy messages to the file named by the pathname *file* without marking the messages as saved. Otherwise, it is equivalent to the **save** command.

EX In the capitalised form, 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, it is equivalent to the **Save** command.

Delete Messages

Synopsis: `d[elete] [msglist]`

Mark messages for deletion from the mailbox. The deletions will not occur until *mailx* quits (see the **quit** command) or changes mailboxes (see the **folder** command). If **autoprint** is set, the next message after the last one deleted will be written; if there is no subsequent message, the previous message, if it exists, will be written. In the case of no subsequent or previous message, or when **noautoprint** is set, the *mailx* prompt will be written.

Discard Header Fields

Synopsis: `di[scard] [header-field...]`
Synopsis: `ig[nore] [header-field...]`

Suppress the specified header fields when writing messages. Specified *header-fields* will be added to the list of suppressed header fields. Examples of header fields to ignore are **status** and **cc**. The fields will be included when the message is saved. The **Print** and **Type** commands override this command. The comparison of header fields is in a case-insensitive manner. If no arguments are specified, write a list of the currently suppressed header fields to standard output; the listing need not reflect the same order of header fields that were entered.

If both **retain** and **discard** commands are given, **discard** commands are ignored.

Delete Messages and Display

Synopsis: `dp [`*`msglist`*`]`
Synopsis: `dt [`*`msglist`*`]`

Delete the specified messages from the mailbox and write the next message after the last one deleted. If there is no subsequent message, the *mailx* prompt will be written.

Echo a String

EX *Synopsis*: `ec[ho] `*`string ...`*

Echo the given strings, equivalent to the shell *echo* utility.

Edit Messages

Synopsis: `e[dit] [`*`msglist`*`]`

Edit the given messages. The messages will be placed in a temporary file and the utility named
EX by the *EDITOR* variable will be invoked to edit the file. The default editor is *ed*.

The **edit** command merely edits the specified messages in a temporary file. It does not modify the contents of those messages in the mailbox.

Exit

Synopsis: `ex[it]`
Synopsis: `x[it]`

Exit from *mailx* without changing the mailbox. No messages will be saved in the **mbox** (see also **quit**).

Change Folder

Synopsis: `fi[le] [`*`file`*`]`
Synopsis: `fold[er] [`*`file`*`]`

Quit (see the **quit** command) from the current file of messages and read in the file named by the pathname *file*. If no argument is given, the name and status of the current mailbox will be written.

Several unquoted special characters are recognised when used as *file* names, with the following substitutions:

% The system mailbox for the invoking user.

%*user* The system mailbox for *user*.

The previous file.

& The current **mbox**.

+*file* The named file in the **folder** directory. (See the **folder** variable.)

The default file is the current mailbox.

Display List of Folders

Synopsis: `folders`

Write the names of the files in the directory set by the **folder** variable.

Follow up Specified Messages

EX *Synopsis*: `fo[llowup] [`*`message`*`]`
Synopsis: `F[ollowup] [`*`msglist`*`]`

In the lower-case form, 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 **save** and **copy** commands and **outfolder**.

In the capitalised form, 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 Save and Copy commands and **outfolder**.

Display Header Summary for Specified Messages

Synopsis: `f[rom] [`*`msglist`*`]`

Write the header summary for the specified messages.

Display Header Summary

Synopsis: `h[eaders] [`*`message`*`]`

Write the page of headers that includes the message specified. The **screen** variable sets the number of headers per page. See also the **z** command.

Help

Synopsis: `hel[p]`
Synopsis: `?`

Write a summary of commands.

Hold Messages

Synopsis: `ho[ld] [`*`msglist`*`]`
Synopsis: `pre[serve] [`*`msglist`*`]`

Mark the messages in *msglist* to be retained in the mailbox when *mailx* terminates. This overrides any commands that might previously have marked the messages to be deleted. During the current invocation of *mailx*, only the **delete**, **dp** or **dt** commands will remove the *preserve* marking of a message.

Execute Commands Conditionally

Synopsis: `i[f] s|r`
`mail-command``s`
`el[se]`
`mail-command``s`
`en[dif]`

Execute commands conditionally, where **if s** will execute the following *mail-command*s, up to an **else** or **endif**, if the program is in Send Mode, and **if r** will cause the *mail-command*s to be executed only in Receive Mode.

List Available Commands

Synopsis: `l[ist]`

Write a list of all commands available. No explanation is given.

Mail a Message

Synopsis: `m[ail]` *address...*

`Mail a message to the specified addresses or aliases.`

Direct Messages to mbox

Synopsis: `mb[ox] [`*msglist*`]`

Arrange for the given messages to end up in the **mbox** save file when *mailx* terminates normally. See *MBOX*. See also the **exit** and **quit** commands.

Process Next Specified Message

Synopsis: `n[ext] [`*message*`]`

Go to the next message matching *message*.

Pipe Message

Synopsis: `pi[pe] [[`*msglist*`]` *command*`]`
Synopsis: `|` `[[`*msglist*`]` *command*`]`

Pipe the messages through the given *command* by invoking the command interpreter specified by *SHELL* with two arguments: **–c** and *command*. (See also *sh* **–c**.) The command must be given as a single argument. Quoting, described previously, can be used to accomplish this. If no arguments are given, the current message will be piped through the command specified by the value of the **cmd** variable. If the **page** variable is set, a form-feed character will be inserted after each message.

Display Message with Headers

Synopsis: `P[rint] [`*msglist*`]`
Synopsis: `T[ype] [`*msglist*`]`

Write the specified messages, including all header lines, to standard output. Override suppression of lines by the **discard**, **ignore** and **retain** commands. If **crt** is set, the messages longer than the number of lines specified by the **crt** variable will be paged through the command specified by the *PAGER* environment variable.

Display Message

Synopsis: `p[rint] [`*msglist*`]`
Synopsis: `t[ype] [`*msglist*`]`

Write the specified messages to standard output. If **crt** is set, the messages longer than the number of lines specified by the **crt** variable will be paged through the command specified by the *PAGER* environment variable.

Quit

Synopsis: `q[uit]`
Synopsis: *end-of-file*

Terminate *mailx*, storing messages that were read in **mbox** (if the current mailbox is the system mailbox and unless **hold** is set), deleting messages that have been explicitly saved (unless **keepsave** is set), discarding messages that have been deleted and saving all remaining messages in the mailbox.

Reply to a Message List

Synopsis: `R[eply] [`*`msglist`*`]`
Synopsis: `R[espond] [`*`msglist`*`]`

Mail a reply message to the sender of each message in the *msglist*. The subject line will be formed by concatenating **Re:**<space> (unless it already begins with that string) and the subject from the first message. If **record** is set to a filename, the response will be saved at the end of that file.

See also the **flipr** variable.

Reply to a Message

Synopsis: `r[eply] [`*`message`*`]`
Synopsis: `r[espond] [`*`message`*`]`

Mail a reply message to all recipients included in the header of the message. The subject line will be formed by concatenating **Re:**<space> (unless it already begins with that string) and the subject from the message. If **record** is set to a filename, the response will be saved at the end of that file.

See also the **flipr** variable.

Retain Header Fields

Synopsis: `ret[ain] [`*`header-field...`*`]`

Retain the specified header fields when writing messages. This command will override all **discard** and **ignore** commands. The comparison of header fields is in a case-insensitive manner. If no arguments are specified, write a list of the currently retained header fields to standard output; the listing need not reflect the same order of header fields that were entered.

Save Messages

Synopsis: `s[ave] [`*`file`*`]`
Synopsis: `s[ave] [`*`msglist`*`] `*`file`*
EX *Synopsis*: `S[ave] [`*`msglist`*`]`

Save the specified messages in the file named by the pathname *file*, or the **mbox** if the *file* argument is omitted. The file will be created if it does not exist; otherwise, the messages will be appended to the file. The message will be deleted from the mailbox when *mailx* terminates unless **keepsave** is set.

EX In the capitalised form, 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** variable.

Set Variables

Synopsis: `se[t] [`*name*`[=[`*string*`]]...] [`*name*`=`*number*`...] [no`*name*`...]`

Define one or more variables called *name*. The variable can be given a null, string or numeric value. Quoting and backslash escapes can occur anywhere in *string*, as described previously, as if the *string* portion of the argument were the entire argument. The forms *name* and *name*= are equivalent to *name*="" for variables that take string values. The **set** command without arguments will write a list of all defined variables and their values. The no*name* form is equivalent to **unset** *name*.

Invoke a Shell

Synopsis: `sh[ell]`

Invoke an interactive command interpreter (see also *SHELL*).

Display Message Size

Synopsis: `si[ze] [`*msglist*`]`

Write the size in bytes of each of the specified messages.

Read mailx Commands From a File

Synopsis: `so[urce]` *file*

Read and execute commands from the file named by the pathname *file* and return to command mode.

Display Beginning of Messages

Synopsis: `to[p] [`*msglist*`]`

Write the top few lines of each of the specified messages. If the **toplines** variable is set, it is taken as the number of lines to write. The default is 5.

Touch Messages

Synopsis: `tou[ch] [`*msglist*`]`

Touch the specified messages. If any message in *msglist* is not specifically deleted nor saved in a file, it will be placed in the **mbox** upon normal termination. See **exit** and **quit**.

Delete Aliases

Synopsis: `una[lias] [`*alias*`]...`

Delete the specified alias names. If a specified alias does not exist, the results are unspecified.

Undelete Messages

Synopsis: `u[ndelete] [`*msglist*`]`

Remove the deleted markings from the specified messages. If **autoprint** is set, the last message of those restored will be written. If *msglist* is not specified, it defaults to the first deleted message following the current message that has not been undeleted if there is one, or the last deleted message preceding the current message that has not been undeleted otherwise.

Unset Variables

Synopsis: `uns[et] name...`

Cause the specified variables to be erased.

Edit Message with Full-screen Editor

Synopsis: `v[isual] [msglist]`

Edit the given messages with a screen editor. The messages are placed in a temporary file, and the utility named by the *VISUAL* variable will be invoked to edit the file. The default editor is *vi*.

The **visual** command merely edits the specified messages in a temporary file. It does not modify the contents of those messages in the mailbox.

Write Messages to a File

Synopsis: `w[rite] [msglist] file`

Write the given messages to the file specified by the pathname *file*, minus the message header. Otherwise, it is equivalent to the **save** command.

Scroll Header Display

Synopsis: `z[+|–]`

Scroll the header display forward (if + is specified or if no option is specified) or backward (if – is specified) one screenful. The number of headers written is set by the **screen** variable.

Invoke Shell Command

Synopsis: `!command`

Invoke the command interpreter specified by *SHELL* with two arguments: **–c** and *command*. (See also *sh* **–c**.) If the **bang** variable is set, each unescaped occurrence of ! in command is replaced with the command executed by the previous ! command or ~! command escape.

Null Command

Synopsis: `# comment`

This null command (comment) will be ignored by *mailx*.

Display Current Message Number

Synopsis: `=`

Write the current message number.

Command Escapes in mailx

The following commands can be entered only from input mode, by beginning a line with the escape character (by default, tilde (~)). See the **escape** variable description for changing this special character. The format for the commands is:

`<ESC><command-char><separator>[<arguments>]`

where the <*separator*> can be zero or more blank characters.

In the following descriptions, the argument *command* (but not *mailx-command)* must be a shell command string. Any string acceptable to the command interpreter specified by the *SHELL*

variable when it is invoked as **–c** *command_string* is valid. The command can be presented as multiple arguments (that is, quoting is not required).

Command escapes that are listed with *msglist* or *mailx-command* arguments are invalid in Send Mode and produce unspecified results.

~! *command*
Invoke the command interpreter specified by *SHELL* with two arguments: **–c** and *command*; and then return to input mode. If the **bang** variable is set, each unescaped occurrence of ! in *command* is replaced with the command executed by the previous ! command or ~! command escape.

~. Simulate end-of-file (terminate message input).

~: *mailx-command*
~_ *mailx-command*
Perform the command-level request.

~? Write a summary of command escapes.

~A This is equivalent to **~i Sign**.

~a This is equivalent to **~i sign**.

~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* for a description of this file.

~e Invoke the editor, as specified by the *EDITOR* environment variable, on the partial message.

~f [*msglist***]**
Forward the specified messages. The specified messages will be inserted into the current message without alteration. This command escape will also insert message headers into the message with field selection affected by the **discard**, **ignore** and **retain** commands.

~F [*msglist***]**
This will be the equivalent of the **~f** command escape, except that all headers will be included in the message, regardless of previous **discard**, **ignore** and **retain** commands.

~h If standard input is a terminal, prompt for a Subject line and the To, Cc and Bcc lists. Other implementation-dependent headers may also be presented for editing. If the field is written with an initial value, it can be edited as if it had just been typed.

~i *string* Insert the value of the named variable, followed by a newline character, into the text of the message. If the string is unset or null, the message will not be changed.

~m [*msglist***]**
Insert the specified messages into the message, prefixing non-empty lines with the string in the **indentprefix** variable. This command escape will also insert message headers into the message, with field selection affected by the **discard**, **ignore** and **retain** commands.

~M [*msglist***]**
This will be the equivalent of the **~m** command escape, except that all headers will be included in the message, regardless of previous **discard**, **ignore** and **retain** commands.

~p Write the message being entered. If the message is longer than **crt** lines (see **Internal Variables in mailx** on page 465), the output will be paginated as described for the *PAGER* variable.

~q Quit (see the **quit** command) from input mode by simulating an interrupt. If the body of the message is not empty, the partial message will be saved in the dead-letter file. See *DEAD* for a description of this file.

~r *file*
~< *file*
~< **!***command*
Read in the file specified by the pathname *file*. If the argument begins with an exclamation-mark (!), the rest of the string is taken as an arbitrary system command; the command interpreter specified by *SHELL* will be invoked with two arguments: **–c** and *command*. The standard output of *command* will be inserted into the message.

~s *string* Set the subject line to *string*.

~t *name*...
Add the given *name*s to the To list.

~v Invoke the full-screen editor, as specified by the *VISUAL* environment variable, on the partial message.

~w *file* Write the partial message, without the header, onto the file named by the pathname *file*. The file will be created or the message will be appended to it if the file exists.

~x Exit as with **~q**, except the message will not be saved in the dead-letter file.

~| *command*
Pipe the body of the message through the given *command* by invoking the command interpreter specified by *SHELL* with two arguments: **–c** and *command*. If the *command* returns a successful exit status, the standard output of the command will replace the message. Otherwise the message will remain unchanged. If the *command* fails, an error message giving the exit status will be written.

EXIT STATUS

When the **–e** option is specified, the following exit values are returned:

0 Mail was found.
>0 Mail was not found or an error occurred.

Otherwise, the following exit values are returned:

0 Successful completion; note that this status implies that all messages were *sent*, but it gives no assurances that any of them were actually *delivered*.
>0 An error occurred.

CONSEQUENCES OF ERRORS

When in input mode (Receive Mode) or Send Mode:

- If an error is encountered processing a command escape (see **Command Escapes in mailx** on page 475), a diagnostic message will be written to standard error, and the message being composed may be modified, but this condition will not prevent the message from being sent.
- Other errors will prevent the sending of the message.

When in command mode:

- Default.

APPLICATION USAGE

Delivery of messages to remote systems requires the existence of communication paths to such systems. These need not exist.

Input lines are limited to {LINE_MAX} bytes, but mailers between systems may impose more severe line-length restrictions. This document does not place any restrictions on the length of messages handled by *mailx*, and for delivery of local messages the only limitations should be the normal problems of available disk space for the target mail file. When sending messages to external machines, applications are advised to limit messages to less than 50 kilobytes because many mail gateways impose message-length restrictions.

The *mailx* utility cannot guarantee support for all character encodings in all circumstances. For example, inter-system mail may be restricted to 7-bit data by the underlying network, 8-bit data need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used.

The format of the system mailbox is intentionally unspecified. Not all systems will implement system mailboxes as flat files, particularly with the advent of multimedia mail messages. Some system mailboxes may be multiple files, others records in a database. The internal format of the messages themselves are specified with the historical format from Version 7, but only after they have been saved in some file other than the system mailbox. This was done so that many historical applications expecting text-file mailboxes will not be broken.

Some new formats for messages can be expected in the future, probably including binary data, bit maps and various multimedia objects. As described here, *mailx* is not prohibited from handling such messages, but it must store them as text files in secondary mailboxes (unless some extension, such as a variable or command-line option, is used to change the stored format). Its method of doing so is implementation-dependent and might include translating the data into text-file-compatible or readable form or omitting certain portions of the message from the stored output.

The **discard** and **ignore** commands are not inverses of the **retain** command. The **retain** command discards all header-fields except those explicitly retained. The **discard** command keeps all header-fields except those explicitly discarded. If headers exist on the retained header list, **discard** and **ignore** commands are ignored.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *ls*, *mail*, *more*, *vi*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an internationalised environment has been described.

The tilde escape (~) has been corrected to (~ |).

Issue 4

Aligned with the ISO/IEC 9945-2:1993 standard.

This utility is now mandatory; it is optional in Issue 3.

NAME

make – maintain, update and regenerate groups of programs (**DEVELOPMENT**)

SYNOPSIS

```
make [-einpqrst][-f makefile]...[ -k| -S][macros=name]...
[target_name...]
```

DESCRIPTION

The *make* utility can be used as a part of software development to update files that are derived from other files. A typical case is one where object files are derived from the corresponding source files. The *make* utility examines time relationships and updates those derived files (called targets) that have modified times earlier than the modified times of the files (called prerequisites) from which they are derived. A description file (makefile) contains a description of the relationships between files, and the commands that must be executed to update the targets to reflect changes in their prerequisites. Each specification, or rule, consists of a target, optional prerequisites and optional commands to be executed when a prerequisite is newer than the target. There are two types of rule:

- inference rules, which have one target name with at least one period (.) and no slash (/)
- target rules, which can have more than one target name.

In addition, *make* has a collection of built-in macros and inference rules that infer prerequisite relationships to simplify maintenance of programs.

To receive exactly the behaviour described in this section, a portable makefile must:

- include the special target **.POSIX**
- omit any special target reserved for implementations (a leading period followed by upper-case letters) that has not been specified by this section.

The behaviour of *make* is unspecified if either or both of these conditions are not met.

OPTIONS

The *make* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–e Cause environment variables, including those with null values, to override macro assignments within makefiles.

–f *makefile*
Specify a different makefile. The argument *makefile* is a pathname of a description file, which is also referred to as the *makefile*. A pathname of – denotes the standard input. There can be multiple instances of this option, and they will be processed in the order specified. The effect of specifying the same option-argument more than once is unspecified.

–i Ignore error codes returned by invoked commands. This mode is the same as if the special target **.IGNORE** were specified without prerequisites.

–k Continue to update other targets that do not depend on the current target if a non-ignored error occurs while executing the commands to bring a target up-to-date.

–n Write commands that would be executed on standard output, but do not execute them. However, lines with a plus sign (+) prefix will be executed. In this mode, lines with an at sign (@) character prefix will be written to standard output.

–p Write to standard output the complete set of macro definitions and target descriptions. The output format is unspecified.

–q Return a zero exit value if the target file is up-to-date; otherwise, return an exit value of 1. Targets will not be updated if this option is specified. However, a command line (associated with the targets) with a plus sign (+) prefix will be executed.

–r Clear the suffix list and do not use the built-in rules.

–S Terminate *make* if an error occurs while executing the commands to bring a target up-to-date. This will be the default and the opposite of **–k**.

–s Do not write command lines or touch messages (see **–t**) to standard output before executing. This mode is the same as if the special target **.SILENT** were specified without prerequisites.

–t Update the modification time of each target as though a *touch target* had been executed. Targets that have prerequisites but no commands (see **Target Rules** on page 483), or that are already up-to-date, will not be touched in this manner. Write messages to standard output for each target file indicating the name of the file and that it was touched. Normally, the command lines associated with each target are not executed. However, a command line with a plus sign (+) prefix will be executed.

If the **–k** and **–S** options are both specified on the command line, by the *MAKEFLAGS* environment variable, or by the **MAKEFLAGS** macro, the last one evaluated will take precedence. The *MAKEFLAGS* environment variable will be evaluated first and the command line will be evaluated second. Assignments to the **MAKEFLAGS** macro will be evaluated as described in **ENVIRONMENT VARIABLES**.

OPERANDS

The following operands are supported:

target_name
Target names, as defined in **EXTENDED DESCRIPTION**. If no target is specified, while *make* is processing the makefiles, the first target that *make* encounters that is not a special target or an inference rule will be used.

macro=name
Macro definitions, as defined in **Macros** on page 485.

If the *target_name* and *macro=name* operands are intermixed on the command line, the results are unspecified.

STDIN

The standard input will be used only if the *makefile* option-argument is –. See **INPUT FILES**.

INPUT FILES

The input file, otherwise known as the makefile, is a text file containing rules, macro definitions and comments.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *make*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other

internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

MAKEFLAGS
This variable is interpreted as a character string representing a series of option characters to be used as the default options. The implementation will accept both of the following formats (but need not accept them when intermixed):

1. The characters are option letters without the leading hyphens or blank character separation used on a command line.
2. The characters are formatted in a manner similar to a portion of the *make* command line: options are preceded by hyphens and blank-character-separated as described in the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The *macro=name* macro definition operands can also be included. The difference between the contents of *MAKEFLAGS* and the command line is that the contents of the variable will not be subjected to the word expansions (see Section 2.6 on page 31) associated with parsing the command line values.

When the command-line options **–f** or **–p** are used, they will take effect regardless of whether they also appear in *MAKEFLAGS*. If they otherwise appear in *MAKEFLAGS*, the result is undefined.

The *MAKEFLAGS* variable will be accessed from the environment before the makefile is read. At that time, all of the options (except **–f** and **–p**) and command-line macros not already included in **MAKEFLAGS** are added to the **MAKEFLAGS** macro. The **MAKEFLAGS** macro will be passed into the environment as an environment variable for all child processes. If the **MAKEFLAGS** macro is subsequently set by the makefile, it replaces the *MAKEFLAGS* variable currently found in the environment.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

EX *PROJECTDIR*
Provide a directory to be used to search for SCCS files not found in the current directory. In all of the following cases, the search for SCCS files will be made in the directory **SCCS** in the identified directory. If the value of *PROJECTDIR* begins with a slash, it is considered an absolute pathname; otherwise, the home directory of a user of that name is examined for a subdirectory **src** or **source**. If such a directory is found, it is used. Otherwise, the value is used as a relative pathname.

If *PROJECTDIR* is not set or has a null value, the search for SCCS files will be made in the directory **SCCS** in the current directory.

The setting of *PROJECTDIR* affects all files listed in the remainder of this utility description for files with a component named **SCCS**.

The value of the *SHELL* environment variable will not be used as a macro and will not be modified by defining the **SHELL** macro in a makefile or on the command line. All other environment variables, including those with null values, are used as macros, as defined in

Macros on page 485.

ASYNCHRONOUS EVENTS

If not already ignored, *make* will trap SIGHUP, SIGTERM, SIGINT and SIGQUIT and remove the current target unless the target is a directory or the target is a prerequisite of the special target **.PRECIOUS** or unless one of the –**n**, –**p** or –**q** options was specified. Any targets removed in this manner will be reported in diagnostic messages of unspecified format, written to standard error. After this cleanup process, if any, *make* will take the standard action for all other signals.

STDOUT

The *make* utility will write all commands to be executed to standard output unless the –**s** option was specified, the command is prefixed with an at sign, or the special target **.SILENT** has either the current target as a prerequisite or has no prerequisites. If *make* is invoked without any work needing to be done, it will write a message to standard output indicating that no action was taken.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None. However, utilities invoked by *make* may create additional files.

EXTENDED DESCRIPTION

The *make* utility attempts to perform the actions required to ensure that the specified targets are up-to-date. A target is considered out-of-date if it is older than any of its prerequisites or if it does not exist. The *make* utility treats all prerequisites as targets themselves and recursively ensures that they are up-to-date, processing them in the order in which they appear in the rule. The *make* utility uses the modification times of files to determine if the corresponding targets are out-of-date.

After *make* has ensured that all of the prerequisites of a target are up-to-date and if the target is out-of-date, the commands associated with the target entry are executed. If there are no commands listed for the target, the target is treated as up-to-date.

Makefile Syntax

A makefile can contain rules, macro definitions (see **Macros** on page 485), and comments. There are two kinds of rules: inference rules and target rules. The *make* utility contains a set of built-in inference rules. If the –**r** option is present, the built-in rules are not used and the suffix list is cleared. Additional rules of both types can be specified in a makefile. If a rule or macro is defined more than once, the value of the rule or macro will be that of the last one specified. Comments start with a number sign (#) and continue until an unescaped newline character is reached.

EX By default, the following files are tried in sequence: **./makefile**, **./Makefile**, **./s.makefile**, **SCCS/s.makefile**, **./s.Makefile** and **SCCS/s.Makefile**.

The –**f** option directs *make* to ignore any of these default files and use the specified argument as a makefile instead. If the – argument is specified, standard input will be used.

The term *makefile* is used to refer to any rules provided by the user, whether in **./makefile** or its variants, or specified by the –**f** option.

The rules in makefiles consist of the following types of lines: target rules, including special targets (see **Target Rules** on page 483); inference rules (see **Inference Rules** on page 486); macro definitions (see **Macros** on page 485); empty lines; and comments. Comments start with a number sign (#) and continue until an unescaped newline character is reached.

When an escaped newline character (one preceded by a **backslash**) is found anywhere in the makefile, it is replaced, along with any leading white space on the following line, with a single space character.

Makefile Execution

Command lines are processed one at a time by writing the command line to the standard output (unless one of the conditions listed below under @ suppresses the writing) and executing the commands in the line. A tab character may precede the command to standard output. Commands will be executed by passing the command line to the command interpreter in the same manner as if the string were the argument to the **XSH** specification *system*( ) function.

The environment for the command being executed will contain all of the variables in the environment of *make*. The macros from the command line to *make* will be added to *make*'s environment. Other implementation-dependent variables may also be added to *make*'s environment. If any command-line macro has been defined elsewhere, the command-line value will overwrite the existing value. If the *MAKEFLAGS* variable is not set in the environment in which *make* was invoked, in the makefile or on the command line, it will be created by *make*, and will contain all options specified on the command line except for the –**f** and –**p** options. It may also contain implementation-dependent options.

By default, when *make* receives a non-zero status from the execution of a command, it terminates with an error message to standard error.

Command lines can have one or more of the following prefixes: a hyphen (–), an at sign (@), or a plus sign (+). These modify the way in which *make* processes the command. When a command is written to standard output, the prefix is not included in the output.

– If the command prefix contains a hyphen, or the –**i** option is present, or the special target **.IGNORE** has either the current target as a prerequisite or has no prerequisites, any error found while executing the command will be ignored.

@ If the command prefix contains an at sign and the command-line –**n** option is not specified, or the –**s** option is present, or the special target **.SILENT** has either the current target as a prerequisite or has no prerequisites, the command will not be written to standard output before it is executed.

+ If the command prefix contains a plus sign, this indicates a command line that will be executed even if –**n**, –**q** or –**t** is specified.

Target Rules

Target rules are formatted as follows:

```
target [target...]: [prerequisite...][;command]
[<tab>command
<tab>command
...]

line that does not begin with <tab>
```

Target entries are specified by a blank-character-separated, non-null list of targets, then a colon, then a blank-character-separated, possibly empty list of prerequisites. Text following a semicolon, if any, and all following lines that begin with a tab character, are command lines to be executed to update the target. The first non-empty line that does not begin with a tab character or # begins a new entry. An empty or blank line, or a line beginning with #, may begin a new entry.

Applications must select target names from the set of characters consisting solely of periods, underscores, digits and alphabetics from the portable character set (see the **XBD** specification, **Section 4.1**, **Portable Character Set**). Implementations may allow other characters in target names as extensions. The interpretation of targets containing the characters % and " is implementation-dependent.

A target that has prerequisites, but does not have any commands, can be used to add to the prerequisite list for that target. Only one target rule for any given target can contain commands.

Lines that begin with one of the following are called *special targets* and control the operation of *make*:

`.DEFAULT` If the makefile uses this special target, it must be specified with commands, but without prerequisites. The commands will be used by *make* if there are no other rules available to build a target.

`.IGNORE` Prerequisites of this special target are targets themselves; this will cause errors from commands associated with them to be ignored in the same manner as specified by the **–i** option. Subsequent occurrences of **.IGNORE** add to the list of targets ignoring command errors. If no prerequisites are specified, *make* will behave as if the **–i** option had been specified and errors from all commands associated with all targets will be ignored.

`.POSIX` This special target must be specified without prerequisites or commands. If it appears before the first non-comment line in the makefile, *make* will process the makefile as specified by this section; otherwise, the behaviour of *make* is unspecified.

`.PRECIOUS` Prerequisites of this special target will not be removed if *make* receives one of the asynchronous events explicitly described in **ASYNCHRONOUS EVENTS**. Subsequent occurrences of **.PRECIOUS** add to the list of precious files. If no prerequisites are specified, all targets in the makefile will be treated as if specified with **.PRECIOUS**.

EX `.SCCS_GET` This special target must be specified without prerequisites. If this special target is included in a makefile, the commands specified with this target replace the default commands associated with this special target. (See **Default Rules** on page 488.) The commands specified with this target are used to get all SCCS files that are not found in the current directory.

When source files are named in a dependency list, *make* treats them just like any other target. Because the source file is presumed to be present in the directory, there is no need to add an entry for it to the makefile. When a target has no dependencies, but is present in the directory, *make* assumes that that file is up-to-date. If, however, an SCCS file named **SCCS/s.***source_file* is found for a target *source_file*, *make* does some additional checking to assure that the target is up-to-date. If the target is missing, or if the SCCS file is newer, *make* automatically issues the commands specified for the **.SCCS_GET** special target to retrieve the most recent version. However, if the target is writable by anyone, *make* does not retrieve a new version.

`.SILENT` Prerequisites of this special target are targets themselves; this causes commands associated with them to not be written to the standard output before they are executed. Subsequent occurrences of **.SILENT** add to the list of targets with silent commands. If no prerequisites are specified, *make* will behave as if the **–s** option had been specified and no commands or touch messages associated with any target will be written to standard output.

`.SUFFIXES` Prerequisites of **.SUFFIXES** are appended to the list of known suffixes and are used in conjunction with the inference rules (see **Inference Rules** on page 486). If **.SUFFIXES** does not have any prerequisites, the list of known suffixes will be cleared. Makefiles must not associate commands with **.SUFFIXES**.

Targets with names consisting of a leading period followed by the upper-case letters **POSIX** and then any other characters are reserved for future standardisation. Targets with names consisting of a leading period followed by one or more upper-case letters are reserved for implementation extensions.

Macros

Macro definitions are in the form:

string1 = **[** *string2***]**

The macro named *string1* is defined as having the value of *string2*, where *string2* is defined as all characters, if any, after the equal sign, up to a comment character (#) or an unescaped newline character. Any blank characters immediately before or after the equal sign will be ignored.

Subsequent appearances of **$(***string1***)** or **${***string1***}** are replaced by *string2*. The parentheses or braces are optional if *string1* is a single character. The macro $$ is replaced by the single character $.

Applications must select macro names from the set of characters consisting solely of periods, underscores, digits and alphabetics from the portable character set (see the **XBD** specification, **Section 4.1**, **Portable Character Set**). A macro name cannot contain an equal sign. Implementations may allow other characters in macro names as extensions.

Macros can appear anywhere in the makefile. Macros in target lines will be evaluated when the target line is read. Macros in command lines will be evaluated when the command is executed. Macros in macro definition lines will not be evaluated until the new macro being defined is used in a rule or command. A macro that has not been defined will evaluate to a null string without causing any error condition.

The forms **$(***string1***[:***subst1***=[***subst2***]])** or **${***string1***[:***subst1***=[***subst2***]]}** can be used to replace all occurrences of *subst1* with *subst2* when the macro substitution is performed. The *subst1* to be replaced is recognised when it is a suffix at the end of a word in *string1* (where a *word*, in this context, is defined to be a string delimited by the beginning of the line, a blank or newline character).

Macro assignments will be accepted from the sources listed below, in the order shown. If a macro name already exists at the time it is being processed, the newer definition will replace the existing definition.

1. Macros defined in *make*'s built-in inference rules.
2. The contents of the environment, including the variables with null values, in the order defined in the environment.
3. Macros defined in the makefiles, processed in the order specified.
4. Macros specified on the command line. It is unspecified whether the internal macros defined in **Internal Macros** on page 487 are accepted from the command line.

If the **–e** option is specified, the order of processing sources items 2 and 3 will be reversed.

The **SHELL** macro is treated specially. It is provided by *make* and set to the pathname of the shell command language interpreter (see *sh*). The *SHELL* environment variable will not affect the value of the **SHELL** macro. If **SHELL** is defined in the makefile or is specified on the

command line, it will replace the original value of the **SHELL** macro, but will not affect the *SHELL* environment variable. Other effects of defining **SHELL** in the makefile or on the command line are implementation-dependent.

Inference Rules

Inference rules are formatted as follows:

```
target:
<tab>command
[<tab>command]
...
line that does not begin with <tab> or #
```

The *target* portion must be a valid target name (see **Target Rules** on page 483) of the form *.s2* or *.s1.s2* (where *.s1* and *.s2* are suffixes that have been given as prerequisites of the **.SUFFIXES** special target and *s1* and *s2* do not contain any slashes or periods.) If there is only one period in the target, it is a single-suffix inference rule. Targets with two periods are double-suffix inference rules. Inference rules can have only one target before the colon.

The makefile must not specify prerequisites for inference rules; no characters other than white space can follow the colon in the first line, except when creating the *empty rule,* described below. Prerequisites are inferred, as described below.

Inference rules can be redefined. A target that matches an existing inference rule will overwrite the old inference rule. An empty rule can be created with a command consisting of simply a semicolon (that is, the rule still exists and is found during inference rule search, but since it is empty, execution has no effect). The empty rule also can be formatted as follows:

```
rule: ;
```

where zero or more blank characters separate the colon and semicolon.

The *make* utility uses the suffixes of targets and their prerequisites to infer how a target can be made up-to-date. A list of inference rules defines the commands to be executed. By default, *make* contains a built-in set of inference rules. Additional rules can be specified in the makefile.

The special target **.SUFFIXES** contains as its prerequisites a list of suffixes that are to be used by the inference rules. The order in which the suffixes are specified defines the order in which the inference rules for the suffixes are used. New suffixes will be appended to the current list by specifying a **.SUFFIXES** special target in the makefile. A **.SUFFIXES** target with no prerequisites will clear the list of suffixes. An empty **.SUFFIXES** target followed by a new **.SUFFIXES** list is required to change the order of the suffixes.

Normally, the user would provide an inference rule for each suffix. The inference rule to update a target with a suffix *.s1* from a prerequisite with a suffix *.s2* is specified as a target *.s2.s1*. The internal macros provide the means to specify general inference rules. (See **Internal Macros** on page 487.)

When no target rule is found to update a target, the inference rules are checked. The suffix of the target (*.s1*) to be built is compared to the list of suffixes specified by the **.SUFFIXES** special targets. If the *.s1* suffix is found in **.SUFFIXES**, the inference rules are searched in the order defined for the first *.s2.s1* rule whose prerequisite file (**$***.s2*) exists. If the target is out-of-date with respect to this prerequisite, the commands for that inference rule are executed.

If the target to be built does not contain a suffix and there is no rule for the target, the single suffix inference rules will be checked. The single-suffix inference rules define how to build a target if a file is found with a name that matches the target name with one of the single suffixes

appended. A rule with one suffix *.s2* is the definition of how to build *target* from *target.s2*. The other suffix (*.s1*) is treated as null.

EX A tilde (~) in the above rules refers to an SCCS file in the current directory. Thus, the rule **.c~.o** would transform an SCCS C-language 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 ~ is a way of changing any file reference into an SCCS file reference.

Libraries

If a target or prerequisite contains parentheses, it will be treated as a member of an archive library. For the *lib***(***member***.o)** expression *lib* refers to the name of the archive library and *member***.o** to the member name. The member must be an object file with the **.o** suffix. The modification time of the expression is the modification time for the member as kept in the archive library. See *ar*. The **.a** suffix refers to an archive library. The *.s2***.a** rule is used to update a member in the library from a file with a suffix *.s2*.

Internal Macros

The *make* utility maintains five internal macros that can be used in target and inference rules. In order to clearly define the meaning of these macros, some clarification of the terms *target rule*, *inference rule*, *target* and *prerequisite* is necessary.

Target rules are specified by the user in a makefile for a particular target. Inference rules are user- or *make*-specified rules for a particular class of target names. Explicit prerequisites are those prerequisites specified in a makefile on target lines. Implicit prerequisites are those prerequisites that are generated when inference rules are used. Inference rules are applied to implicit prerequisites or to explicit prerequisites that do not have target rules defined for them in the makefile. Target rules are applied to targets specified in the makefile.

Before any target in the makefile is updated, each of its prerequisites (both explicit and implicit) will be updated. This is accomplished by recursively processing each prerequisite. Upon recursion, each prerequisite becomes a target itself. Its prerequisites in turn are processed recursively until a target is found that has no prerequisites, at which point the recursion stops. The recursion then backs up, updating each target as it goes.

In the definitions that follow, the word *target* refers to one of:

- a target specified in the makefile
- an explicit prerequisite specified in the makefile that becomes the target when *make* processes it during recursion
- an implicit prerequisite that becomes a target when *make* processes it during recursion.

In the definitions that follow, the word *prerequisite* refers to one o the following:

- an explicit prerequisite specified in the makefile for a particular target
- an implicit prerequisite generated as a result of locating an appropriate inference rule and corresponding file that matches the suffix of the target.

The five internal macros are:

$@ The $@ evaluates to the full target name of the current target, or the archive filename part of a library archive target. It is evaluated for both target and inference rules.

For example, in the **.c.a** inference rule, $@ represents the out-of-date **.a** file to be built. Similarly, in a makefile target rule to build **lib.a** from **file.c**, $@ represents the out-of-date **lib.a**.

$% The $% macro is evaluated only when the current target is an archive library member of the form *libname*(*member***.o**). In these cases, $@ evaluates to *libname* and $% evaluates to *member***.o**. The $% macro is evaluated for both target and inference rules.

For example, in a makefile target rule to build **lib.a(file.o)**, $% represents **file.o** as opposed to $@, which represents **lib.a**.

$? The $? macro evaluates to the list of prerequisites that are newer than the current target. It is evaluated for both target and inference rules.

For example, in a makefile target rule to build **prog** from **file1.o**, **file2.o** and **file3.o**, and where **prog** is not out of date with respect to **file1.o**, but is out of date with respect to **file2.o** and **file3.o**, $? represents **file2.o** and **file3.o**.

$< In an inference rule, $< evaluates to the file name whose existence allowed the inference rule to be chosen for the target. In the **.DEFAULT** rule, the $< macro evaluates to the current target name. The $< macro is evaluated only for inference rules.

For example, in the **.c.a** inference rule, $< represents the prerequisite **.c** file.

$* The $* macro evaluates to the current target name with its suffix deleted. It is evaluated at least for inference rules.

For example, in the **.c.a** inference rule, $*.o represents the out-of-date **.o** file that corresponds to the prerequisite **.c** file.

Each of the internal macros has an alternative form. When an upper-case D or F is appended to any of the macros, the meaning is changed to the *directory part* for D and *filename part* for F. The directory part is the path prefix of the file without a trailing slash; for the current directory, the directory part is ‘‘.’’. When the $? macro contains more than one prerequisite filename, the $(?D) and $(?F) (or ${?D} and ${?F}) macros expand to a list of directory name parts and filename parts respectively.

For the target *lib***(***member***.o)** and the **s2.a** rule, the internal macros are defined as:

$< *member***.s2**

$* *member*

$@ *lib*

$? *member***.s2**

$% *member***.o**

Default Rules

The default rules for *make* achieve results that are the same as if the following were used. Implementations that do not support FORTRAN may omit **FC**, **FFLAGS** and the .f inference rules. Implementations may provide additional macros and rules.

SPECIAL TARGETS

EX
```
.SCCS_GET: sccs $(SCCSFLAGS) get $(SCCSGETFLAGS) $@
```

EX `.SUFFIXES: .o .c .y l .a .sh .f .c~ .y~ .l~ .sh~ .f~`

MACROS:

```
MAKE=make
AR=ar
ARFLAGS=-rv
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LDFLAGS=
CC=c89
CFLAGS=-O
FC=fort77
FFLAGS=-O 1
```

EX
```
GET=get
GFLAGS=
SCCSFLAGS=
SCCSGETFLAGS=-s
```

SINGLE SUFFIX RULES

```
.c:
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<

.f:
	$(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<

.sh:
	cp $< $@
	chmod a+x $@
```

EX
```
.c~:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $*.c

.f~:
	$(GET) $(GFLAGS) -p $< > $*.f
	$(FC) $(FFLAGS) $(LDFLAGS) -o $@ $*.f

.sh~:
	$(GET) $(GFLAGS) -p $< > $*.sh
	cp $*.sh $@
	chmod a+x $@
```

DOUBLE SUFFIX RULES

```
.c.o:
	$(CC) $(CFLAGS) -c $<

.f.o:
	$(FC) $(FFLAGS) -c $<
```

```
.y.o:
	$(YACC) $(YFLAGS) $<
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c
	mv y.tab.o $@

.l.o:
	$(LEX) $(LFLAGS) $<
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c
	mv lex.yy.o $@

.y.c:
	$(YACC) $(YFLAGS) $<
	mv y.tab.c $@

.l.c:
	$(LEX) $(LFLAGS) $<
	mv lex.yy.c $@
```

EX

```
.c~.o:
	$(GET) $(GFLAGS) -p $< > $*.c
	$(CC) $(CFLAGS) -c $*.c

.f~.o:
	$(GET) $(GFLAGS) -p $< > $*.f
	$(FC) $(FFLAGS) -c $*.f

.y~.o:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	$(CC) $(CFLAGS) -c y.tab.c
	rm -f y.tab.c
	mv y.tab.o $@

.l~.o:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(LFLAGS) $*.l
	$(CC) $(CFLAGS) -c lex.yy.c
	rm -f lex.yy.c
	mv lex.yy.o $@

.y~.c:
	$(GET) $(GFLAGS) -p $< > $*.y
	$(YACC) $(YFLAGS) $*.y
	mv y.tab.c $@

.l~.c:
	$(GET) $(GFLAGS) -p $< > $*.l
	$(LEX) $(LFLAGS) $*.l
	mv lex.yy.c $@
```

```
.c.a:
	$(CC) -c $(CFLAGS) $<
	$(AR) $(ARFLAGS) $@ $*.o
	rm -f $*.o
```

```
.f.a:
        $(FC) -c $(FFLAGS) $<
        $(AR) $(ARFLAGS) $@ $*.o
        rm -f $*.o
```

EXIT STATUS

When the –**q** option is specified, the *make* utility will exit with one of the following values:

0 Successful completion.
1 The target was not up-to-date.
>1 An error occurred.

When the –**q** option is not specified, the *make* utility will exit with one of the following values:

0 successful completion
>0 an error occurred

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If there is a source file (such as **./source.c**) and there are two SCCS files corresponding to it (**./s.source.c** and **./SCCS/s.source.c**), *make* will use the SCCS file in the current directory. However, users are advised to use the underlying SCCS utilities (*admin*, *delta*, *get*, and so forth) or the *sccs* utility for all source files in a given directory. If both forms are used for a given source file, future developers may be confused.

It is incumbent upon portable makefiles to specify the **.POSIX** special target in order to guarantee that they are not affected by local extensions.

The –**k** and –**S** options are both present so that the relationship between the command line, the *MAKEFLAGS* variable, and the makefile can be controlled precisely. If the k flag is passed in *MAKEFLAGS* and a command is of the form:

```
$(MAKE) -S foo
```

then the default behaviour is restored for the child *make*.

When the –**n** option is specified, it is always added to *MAKEFLAGS*. This allows a recursive *make* –**n** *target* to be used to see all of the action that would be taken to update *target*.

Because of widespread historical practice, interpreting a # number sign inside a variable as the start of a comment has the unfortunate side effect of making it impossible to place a number sign in a variable, thus forbidding something like:

```
CFLAGS = "-D COMMENT_CHAR='#'"
```

Many historical *make* utilities stop chaining together inference rules when an intermediate target is non-existent. For example, it might be possible for a *make* to determine that both .y.c and .c.o could be used to convert a .y to a .o. Instead, in this case, *make* requires the use of a .y.o rule.

The best way to provide portable makefiles is to include all of the rules needed in the makefile itself. The rules provided use only features provided by other parts of the standard. The default rules include rules for optional commands in the standard. Only rules pertaining to commands that are provided are needed in an implementation's default set.

Macros used within other macros are evaluated when the new macro is used rather than when the new macro is defined. Therefore:

```
MACRO = value1
NEW   = $(MACRO)
MACRO = value2

target:
        echo $(NEW)
```

would produce *value2* and not *value1* since **NEW** was not expanded until it was needed in the *echo* command line.

Some historical applications have been known to intermix *target_name* and *macro=name* operands on the command line, expecting that all of the macros will be processed before any of the targets are dealt with. Portable applications do not do this, although some backward compatibility support may be included in some implementations.

The following characters in filenames may give trouble:

```
=   :   ‘   ’   @
```

For inference rules, the description of $< and $? seem similar. However, an example shows the minor difference. In a makefile containing:

```
foo.o: foo.h
```

if **foo.h** is newer than **foo.o**, yet **foo.c** is older than **foo.o**, the built-in rule to make **foo.o** from **foo.c** will be used, with $< equal to **foo.c** and $? equal to **foo.h**. If **foo.c** is also newer than **foo.o**, $< is equal to **foo.c** and $? is equal to **foo.h foo.c**.

EXAMPLES

1. The following command:

   ```
   make
   ```

   makes the first target found in the makefile.

2. The following command:

   ```
   make junk
   ```

   makes the target **junk**.

3. 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
          c89 a.o b.o –o pgm

   a.o: incl.h a.c
          c89 –c a.c

   b.o: incl.h b.c
          c89 –c b.c
   ```

4. An example for making optimised **.o** files from **.c** files is:

   ```
   .c.o:
          c89 –c –O $*.c
   ```

or:

```
.c.o:
        c89 –c –O $<
```

5. The most common use of the archive interface follows. Here, it is assumed that the source files are all C-language source:

```
lib:    lib(file1.o) lib(file2.o) lib(file3.o)
        @echo lib is now up-to-date
```

The **.c.a** rule is used to make **file1.o**, **file2.o** and **file3.o** and insert them into **lib**.

The treatment of escaped newline characters throughout the makefile is historical practice. For example, the inference rule:

```
.c.o\

:
```

works, and the macro:

```
f=      bar baz\

        biz

a:
        echo ==$f==
```

will echo ==bar baz biz==.

If $? were:

```
/usr/include/stdio.h /usr/include/unistd.h foo.h
```

then $(?D) would be:

```
/usr/include /usr/include .
```

and $(?F) would be:

```
stdio.h unistd.h foo.h
```

6. The contents of the built-in rules can be viewed by running:

```
make -p -f /dev/null 2>/dev/null
```

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *c89*, *cc*, *get*, *lex*, *sh*, *yacc*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Issue 4, Version 2

Under **Default Rules**, the string `-G$@` is deleted from the line referencing `sccs`.

NAME

man – display system documentation

SYNOPSIS

`man [-k] name...`

DESCRIPTION

The *man* utility writes information about each of the *name* operands. If *name* is the name of a standard utility, *man* will at a minimum write a message describing the syntax used by the standard utility, its options and operands. If more information is available, the *man* utility will provide it in an implementation-dependent manner.

An implementation may provide information for values of *name* other than the standard utilities. Standard utilities that are listed as optional and that are not supported by the implementation either will cause a brief message indicating that fact to be displayed or will cause a full display of information as described previously.

OPTIONS

The *man* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–k Interpret *name* operands as keywords to be used in searching a utilities summary database that contains a brief purpose entry for each standard utility and write lines from the summary database that match any of the keywords. The keyword search produces results that are the equivalent of the output of the following command:

```
grep -Ei '
name
name
...
' summary-database
```

This assumes that the *summary-database* is a text file with a single entry per line; this organisation is not required and the example using *grep* –**Ei** is merely illustrative of the type of search intended. The purpose entry to be included in the database consists of a terse description of the purpose of the utility.

OPERANDS

The following operand is supported:

name A keyword or the name of a standard utility. When –**k** is not specified and *name* does not represent one of the standard utilities, the results are unspecified.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *man*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and in the summary database). The value of *LC_CTYPE* need not affect the format of the information written about the name operands.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PAGER

Determine an output filtering command for writing the output to a terminal. Any string acceptable as a *command_string* operand to the *sh* **–c** command is valid. When standard output is a terminal device, the manual-page output will be piped through the command. If the *PAGER* variable is null or not set, the command will be either *more* or another paginator utility documented in the system documentation.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *man* utility writes text describing the syntax of the utility *name*, its options and its operands, or, when **–k** is specified, lines from the summary database. The format of this text is implementation-dependent.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

more.

CHANGE HISTORY

First released in Issue 4.

NAME

mesg – permit or deny messages

SYNOPSIS

```
mesg [y|n]
```

DESCRIPTION

The *mesg* utility will control whether other users are allowed to send messages via *write*, *talk* or other utilities to a terminal device. The terminal device affected is determined by searching for the first terminal in the sequence of devices associated with standard input, standard output and standard error, respectively. With no arguments, *mesg* reports the current state without changing it. Processes with appropriate privileges may be able to send messages to the terminal independent of the current state.

OPTIONS

None.

OPERANDS

The following operands are supported in the POSIX locale:

y Grant permission to other users to send messages to the terminal device.

n Deny permission to other users to send messages to the terminal device.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *mesg*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written (by *mesg*) to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If no operand is specified, *mesg* displays the current terminal state in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Receiving messages is allowed.
1 Receiving messages is not allowed.
>1 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The mechanism by which the message status of the terminal is changed is unspecified. Therefore, unspecified actions may cause the status of the terminal to change after *mesg* has successfully completed. These actions may include, but are not limited to: another invocation of the *mesg* utility; login procedures; invocation of the *stty* utility; invocation of the *chmod* utility or *chmod*( ) function; and so forth.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

talk, *write*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

mkdir – make directories

SYNOPSIS

```
mkdir [-p][-m mode] dir...
```

DESCRIPTION

The *mkdir* utility will create the directories specified by the operands, in the order specified.

For each *dir* operand, the *mkdir* utility will perform actions equivalent to the **XSH** specification *mkdir*( ) function, called with the following arguments:

1. The *dir* operand is used as the *path* argument.
2. The value of the bitwise inclusive OR of S_IRWXU, S_IRWXG and S_IRWXO is used as the *mode* argument. (If the –**m** option is specified, the *mode* option-argument overrides this default.)

OPTIONS

The *mkdir* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**m** *mode*
: Set the file permission bits of the newly-created directory to the specified *mode* value. The *mode* option-argument will be the same as the *mode* operand defined for the *chmod* utility. In the *symbolic_mode* strings, the *op* characters + and – will be interpreted relative to an assumed initial mode of a=rwx; + will add permissions to the default mode, – will delete permissions from the default mode.

–**p**
: Create any missing intermediate pathname components.

  For each *dir* operand that does not name an existing directory, effects equivalent to those caused by the following command will occur:

```
mkdir -p -m $(umask -S),u+wx $(dirname dir) &&
mkdir [–m mode] dir
```

  where the **[**–**m** *mode***]** option represents that option supplied to the original invocation of *mkdir*, if any.

  Each *dir* operand that names an existing directory will be ignored without error.

OPERANDS

The following operand is supported:

dir
: A pathname of a directory to be created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *mkdir*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All the specified directories were created successfully or the **–p** option was specified and all the specified directories now exist.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The default file mode for directories is a=rwx (777 on most systems) with selected permissions removed in accordance with the file mode creation mask. For intermediate pathname components created by *mkdir*, the mode is the default modified by u+wx so that the subdirectories can always be created regardless of the file mode creation mask; if different ultimate permissions are desired for the intermediate directories, they can be changed afterwards with *chmod*.

Note that some of the requested directories may have been created even if an error occurs.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
rm, *rmdir*, *umask*, the **XSH** specification description of *mkdir*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

mkfifo – make FIFO special files

SYNOPSIS

```
mkfifo [-m mode] file...
```

DESCRIPTION

The *mkfifo* utility will create the FIFO special files specified by the operands, in the order specified.

For each *file* operand, the *mkfifo* utility will perform actions equivalent to the **XSH** specification *mkfifo*( ) function, called with the following arguments:

1. The *file* operand is used as the *path* argument.
2. The value of the bitwise inclusive OR of S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH and S_IWOTH is used as the *mode* argument. (If the **–m** option is specified, the *mode* option-argument overrides this default.)

OPTIONS

The *mkfifo* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–m *mode*
: Set the file permission bits of the newly-created FIFO to the specified *mode* value. The *mode* option-argument will be the same as the *mode* operand defined for the *chmod* utility. In the *symbolic_mode* strings, the *op* characters + and – will be interpreted relative to an assumed initial mode of a=rw.

OPERANDS

The following operand is supported:

file A pathname of the FIFO special file to be created.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *mkfifo*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All the specified FIFO special files were created successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
umask, the **XSH** specification description of *mkfifo*( ).

CHANGE HISTORY
First released in Issue 3.

Issue 4
Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

more – display files on a page-by-page basis

SYNOPSIS

```
more [-ceisu][-n number][-p command][-t tagstring][file...]
```

OB

```
more [-ceisu][-n number][+command][-t tagstring][file...]
```

DESCRIPTION

The *more* utility reads files and either writes them to the terminal on a page-by-page basis or filters them to standard output. If standard output is not a terminal device, all input files are copied to standard output in their entirety, without modification. If standard output is a terminal device, the files will be written a number of lines (one screenful) at a time under the control of user commands; see **EXTENDED DESCRIPTION**.

Certain block-mode terminals do not have all the capabilities necessary to support the complete *more* definition; they are incapable of accepting commands that are not terminated with a newline character. Implementations that support such terminals provide an operating mode to *more* in which all commands can be terminated with a newline character on those terminals. This mode will:

- be documented in the system documentation
- at invocation, inform the user of the terminal deficiency that requires the newline character usage and provide instructions on how this warning can be suppressed in future invocations
- not be required for implementations supporting only fully capable terminals
- not affect commands already requiring newline characters
- not affect users on the capable terminals from using *more* as described in this document

OPTIONS

OB The *more* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that *+command* of the obsolescent version uses a non-standard syntax.

The following options are supported:

–c If a screen is to be written that has no lines in common with the current screen, or *more* is writing its first screen, *more* will not scroll the screen, but instead will redraw each line of the screen in turn, from the top of the screen to the bottom. In addition, if *more* is writing its first screen, the screen will be cleared. This option may be ignored on devices that do not have the ability to clear to the end of a line or the end of a screen.

–e Exit immediately after writing the last line of the last file in the argument list; see **EXTENDED DESCRIPTION**.

–i Perform pattern matching in searches without regard to case. See the **XBD** specification, **Section 7.2**, **Regular Expression General Requirements** .

–n *number*

Specify the number of lines per screenful. The *number* argument is a positive decimal integer. The **–n** option overrides any values obtained from the environment.

–p *command*

OB *+command*

For each file examined, initially execute the *more* command in the *command* argument. If the command is a positioning command, such as a line number or a regular expression search, set the current position (see **EXTENDED DESCRIPTION**) to represent the final results of the command, without writing any intermediate lines of

the file. For example, the two commands:

```
more -p 1000j file
more -p 1000G file
```

would be equivalent and start the display with the current position at line 1000, bypassing the lines that **j** would write and scroll off the screen if it had been issued during the file examination. If the positioning command is unsuccessful, the first line in the file will be the current position.

–s Replace consecutive empty lines with a single empty line.

–t *tagstring*
Write the screenful of the file containing the tag named by the *tagstring* argument. See the *ctags* utility. The tags feature represented by **–t** *tagstring* and the **:t** command is optional. It is provided on any system that also provides a conforming implementation of *ctags*; otherwise, the use of **–t** produces undefined results.

–u Treat a backspace character as a printable control character, displayed as an implementation-dependent character sequence (see **EXTENDED DESCRIPTION**), suppressing backspacing and the special handling that produces underlined or standout-mode text on some terminal types. Also, do not ignore a carriage-return character at the end of a line.

OB If both the **–t** *tagstring* and **–p** *command* (or the obsolescent **+***command*) options are given, the **–t** *tagstring* will be processed first; that is, the file containing the tag is selected by **–t** and then the command is executed.

OPERANDS

The following operand is supported:

file A pathname of an input file. If no *file* operands are specified, the standard input will be used. If a *file* is –, the standard input will be read at that point in the sequence.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –.

INPUT FILES

The input files being examined must be text files. If standard output is a terminal, standard error will be used to read commands from the user. If standard output is a terminal, standard error is not readable, and command input is needed, *more* will terminate with an error indicating that it was unable to read user commands. If standard output is not a terminal, no error will result if standard error cannot be opened for reading.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *more*:

COLUMNS
Override the system-selected horizontal screen size. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

EDITOR
Used by the **v** command to select an editor; see **EXTENDED DESCRIPTION**.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within regular expressions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

LINES Override the system-selected vertical screen size, used as the number of lines in a screenful. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null. The **–n** option takes precedence over the *LINES* variable for determining the number of lines in a screenful.

MORE Determine a string containing options described in **OPTIONS**, preceded with hyphens and blank-character-separated as on the command line. Any command-line options are processed after those in the *MORE* variable, as if the command line were:

```
more $MORE options operands
```

The *MORE* variable takes precedence over the *TERM* and *LINES* variables for determining the number of lines in a screenful.

TERM Determine the name of the terminal type. If this variable is unset or null, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output will be used to write the contents of the input files.

STDERR
Used for diagnostic messages and user commands (see **INPUT FILES**), and, if standard output is a terminal device, to write a prompting string. The prompting string will only appear as the last line of the screen and will contain the name of the file currently being examined, but it is otherwise unspecified. User input for the **/**, **?**, **:e** and **:t** commands will be written on the same line of the screen as the prompt. It is unspecified if informational messages are written for other user commands.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
The number of lines available per screen is determined by the **–n** option, if present or by examining values in the environment (see **ENVIRONMENT VARIABLES**). If neither method yields a number, an unspecified number of lines will be used. The actual number of lines written

will be one less than this number, as the last line of the screen will be used to write a user prompt and user input. If the number of lines available per screen is less than four, the results are undefined.

In the following descriptions, the *current position* refers to two things:

- the position of the current line on the screen
- the line number (in the file) of the current line on the screen.

Usually, the line on the screen corresponding to the current position is the third line on the screen. If this is not possible (there are fewer than three lines to display or this is the first page of the file, or it is the last page of the file), then the current position is either the first or last line on the screen as described later.

The number of columns available per line is determined by examining values in the environment (see **ENVIRONMENT VARIABLES**), with a default value as described in the **XBD** specification, **Chapter 6**, **Environment Variables**. The *more* utility writes lines containing more characters than would fit into this number of columns by breaking the line into one or more logical lines where each of these lines but the last contains the number of characters needed to fill the columns. The logical lines are written independently of each other; that is, commands affecting a single line affect them separately.

When standard output is a terminal and **–u** is not specified, *more* treats backspace characters and carriage-return characters specially:

- A character, followed first by a backspace character, then by an underscore (_), will cause that character to be written as underlined text, if the terminal type supports that. An underscore, followed first by a backspace character, then any character, will also cause that character to be written as underlined text, if the terminal type supports that.
- A backspace character that appears between two identical printable characters will cause the first of those two characters to be written as emboldened text (that is, visually brighter, standout mode or inverse-video mode), if the terminal type supports that, and the second to be discarded. Immediately subsequent occurrences of backspaces/character pairs for that same character will also be discarded. (For example, the sequence a\ba\ba\ba is interpreted as a single emboldened a.)
- Other backspace character sequences will be written directly to the terminal, which generally cause the character preceding the backspace character to be suppressed in the display.
- A carriage-return character at the end of a line will be ignored, rather than being written as a control character, as described in the next paragraph.

It is implementation-dependent how other non-printable characters are written. Implementations should use the same format that they use for the *ex* **print** command.

If the **–t** option was specified, *more* will write a screen of the file containing the specified tag in the current position.

If the **–p** option was specified, *more* will execute the command supplied in the *command* option-argument, and write the screen so that the current position is the current position that would result from applying that command to the input file.

If neither the **–p** or **–t** options were specified, *more* will write the first screen of the first input file.

Once the initial screen has been written, *more* will prompt the user and, based on the interactive user input, will modify the screen or exit. If the modification of the screen results in a screen that has lines in common with the current screen, *more* will scroll the screen rather than clearing and redrawing the screen (unless the **–c** option is specified). If the modification of the screen results

in a screen that has no lines in common with the current screen, *more* will clear and redraw the screen.

If the standard output is not a terminal device, *more* will always exit when it reaches end-of-file on the last file in its argument list. Otherwise, for all files but the last, *more* will prompt, with an indication that it has reached the end of file, along with the name of the next file. For the last file specified, or for the standard input if no file is specified, *more* will prompt, indicating end-of-file, and accept additional commands. If the next command specifies forward scrolling, *more* will exit. If the **–e** option is specified, *more* will exit immediately after writing the last line of the last file.

Several of the commands described in this section move the current position backwards in the input stream. In the case that text is being taken from a non-rewindable stream, such as a pipe, it is implementation-dependent how much backward motion is supported.

If a scrolling command cannot be performed because there are insufficient lines to scroll, *more* will alert the terminal. The scrolling commands are b, <control>-B, d, <control>-D, f, <control>-F, g, G, j, k, s, u and <control>-U.

The interactive commands in the following sections are supported. Some commands can be preceded by a decimal integer, called *count* in the following descriptions. If not specified with the command, *count* defaults to 1.

In the following descriptions, *pattern* is a basic regular expression, as described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**. The term ‘‘examine’’ is historical usage meaning ‘‘open the file for viewing’’; for example, *more* **foo** would be expressed as examining file **foo**.

Help

Synopsis: h

Write a summary of these commands and other implementation-dependent commands.

Move Forward One Screenful

Synopsis: **[***count***]**f
Synopsis: **[***count***]**<control>-F

Move forward *count* lines, with a default of one screenful. At end-of-file, *more* will continue with the next file in the list, or exit if the current file is the last file in the list.

Move Backward One Screenful

Synopsis: **[***count***]**b
Synopsis: **[***count***]**<control>-B

Move backward *count* lines, with a default of one screenful (see option **–n**). If *count* is more than the screen size, only the final screenful will be written.

Scroll Forward One Line

Synopsis: **[***count***]**<space>
Synopsis: **[***count***]**j
Synopsis: **[***count***]**<newline>

Scroll forward *count* lines. The default *count* for the space character will be one screenful; for j and newline character, one line. The entire *count* lines will be written, even if *count* is more than the screen size. At end-of-file, a newline character will cause *more* to continue with the next file in the list, or exit if the current file is the last file in the list.

Scroll Backward One Line

Synopsis: **[***count***]**k

Scroll backward *count* lines, with a default of 1. The entire *count* lines will be written, even if *count* is more than the screen size.

Scroll Forward One Half Screenful

Synopsis: **[***count***]**d
Synopsis: **[***count***]**<control>-D

Scroll forward *count* lines, with a default of one half of the screen size. If *count* is specified, it will become the new default for subsequent **d** and **u** commands.

Skip Forward One Line

Synopsis: **[***count***]**s

Skip forward *count* lines, with a default of 1, and write the next screenful beginning at that point. If *count* would cause the current position to be such that less than one screenful would be written, the last screenful in the file will be written.

Scroll Backward One Half Screenful

Synopsis: **[***count***]**u
Synopsis: **[***count***]**<control>-U

Scroll backward *count* lines, with a default of one half of the screen size. If *count* is specified, it will become the new default for subsequent **d** and **u** commands.

Go to Beginning of File

Synopsis: **[***count***]**g

Go to line *count* in the file, with a default of 1 (beginning of file). Scroll or rewrite the screen so that that line is at the current position.

Go to End of File

Synopsis: **[***count***]**G

Go to line *count* in the file, with a default of the end of the file. If no *count* is specified, scroll or rewrite the screen so that the last line in the file is at the bottom of the screen. If *count* is specified, scroll or rewrite the screen so that that line is at the current position.

Refresh the Screen

Synopsis: `r`
Synopsis: `<control>-L`

Refresh the screen.

Discard and Refresh

Synopsis: `R`

Refresh the screen, discarding any buffered input. If the current file is non-seekable, buffered input will not be discarded and the **R** command is equivalent to the **r** command.

Mark Position

Synopsis: `m`*`letter`*

Mark the current position with the letter named by *letter*, where *letter* represents the name of one of the lower-case letters of the portable character set. When a new file is examined, all marks may be lost.

Return to Mark

Synopsis: `'`*`letter`*

Return to the position that was previously marked with the letter named by *letter*, making that line the current position.

Return to Previous Position

Synopsis: `''`

Return to the position from which the last large movement command was executed (where a ''large movement'' is defined as any movement of more than a screenful of lines). If no such movements have been made, return to the beginning of the file.

Search Forward for Pattern

Synopsis: `[`*`count`*`]/[!]`*`pattern`*`<newline>`

Search forward in the file for the *count*th line containing the *pattern*. The *count* defaults to 1. The search will start at the line following the current position. If the search is successful, the screen will be modified so that the searched-for line is in the current position. The null regular expression (/ followed by a newline character) will repeat the search using the previous regular expression. If the character ! is included, the lines for searching will be those that do not contain the *pattern*.

If a match is found for the *pattern*, the logical line containing the pattern will be written in the current position. If the matching line is already on the screen, the screen will be scrolled to make that line the current position; otherwise, a full screen will be written. If no match is found for the pattern, a message to that effect will be written as a part of the prompt, and the screen and the current position will remain unchanged. However, if no match is found and the input is the standard input, the screen may be scrolled to the last screenful of the input.

Search Backward for Pattern

Synopsis: **[***count***]?[!]***pattern*<newline>

Search backward in the file for the *count*th line containing the *pattern*. The search will start at the line immediately before the current position. If the search is successful, the screen will be modified so that the searched-for line is in the current position. The null regular expression (? followed by a newline character) will repeat the search using the previous regular expression. If the character ! is included, the lines for searching will be those that do not contain the *pattern*.

If a match is found for the *pattern*, the logical line containing the pattern will be written in the current position. If the matching line is already on the screen, the screen will be scrolled to make that line the current position; otherwise, a full screen will be written. If no match is found for the pattern, a message to that effect will be written as a part of the prompt, and the screen and the current position will remain unchanged. However, if no match is found and the input is the standard input, the screen may be scrolled to the last screenful of the input.

Repeat Search

Synopsis: **[***count***]**n

Repeat the previous search for *count*th line (default 1) containing the last *pattern* (or not containing the last *pattern*, if the previous search was /! or ?!).

Repeat Search in Reverse

Synopsis: **[***count***]**N

Repeat the search in the opposite direction of the previous search for the *count*th line (default 1) containing the last *pattern* (or not containing the last *pattern*, if the previous search was /! or ?!).

Examine New File

Synopsis: :e **[***filename***]**<newline>

Examine a new file. If the *filename* argument is not specified, the current file (see the **:n** and **:p** commands below) from the list of files in the command line will be reexamined. The *filename* will be subjected to the process of shell word expansions (see Section 2.6 on page 31); if more than a single pathname results, the effects are unspecified. If *filename* is a number sign (#), the previously examined file will be reexamined. If *filename* refers to a non-seekable file, the results are unspecified.

Examine Next File

Synopsis: **[***count***]**:n

Examine the next file. If a number *count* is specified, the *count*th next file will be examined. If *filename* refers to a non-seekable file, the results are unspecified.

Examine Previous File

Synopsis: **[***count***]**:p

Examine the previous file. If a number *count* is specified, the *count*th previous file will be examined. If *filename* refers to a non-seekable file, the results are unspecified.

Go to Tag

Synopsis: `:t` *tagstring*`<newline>`

Go to the supplied *tagstring* and scroll/rewrite the screen with that line in the current position; see the **–t** option. If the *ctags* utility is not supported by the system, the use of :t produces undefined results.

Invoke Editor

Synopsis: `v`

Invoke an editor to edit the current file being examined. If standard input is being examined, the results are unspecified. The name of the editor will be taken from the environment variable *EDITOR*, or defaults to *vi*. If *EDITOR* represents either *vi* or *ex*, the editor will be invoked with options such that the current editor line is the physical line corresponding to the current position in *more* at the time of invocation. For example, either *ex* or *vi* is invoked by specifying the editor name and following that with **–c** *linenumber*. It is implementation-dependent whether line-setting options are passed to editors other than *vi* and *ex*.

The file types that can be edited are implementation-dependent.

When the editor exits, *more* will resume on the current file by rewriting the screen with the current line as the current position.

Display Position

Synopsis: `=`
Synopsis: `<control>-G`

Write the name of the file currently being examined, the number relative to the total number of files there are to examine, the current line number, the current byte number and the total bytes to write and what percentage of the file precedes the current position. If *more* is reading from standard input, or the file is shorter than a single screen, some of these items need not be written. All of these items will reference the first byte of the line after the last line written.

Quit

Synopsis: `q`
Synopsis: `:q`
Synopsis: `ZZ`

Exit *more*.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If an error is encountered accessing a file when using the **:n** command, *more* will attempt to examine the next file in the argument list, but the final exit status will be affected. If an error is encountered accessing a file via the **:p** command, *more* will attempt to examine the previous file in the argument list, but the final exit status will be affected. If an error is encountered accessing a file via the **:e** command, *more* will remain in the current file and the final exit status will not be affected.

APPLICATION USAGE

The operating mode referred to for block-mode terminals effectively adds a newline character to each synopsis line that currently has none. So, for example, **d**<newline> would page one screenful. The mode could be triggered by a command-line option, environment variable or some other method.

When the standard output is not a terminal, none of the filter-modification options are effective. This is based on historical practice. For example, a typical implementation of *man* pipes its output through *more* –**s** to squeeze excess white space for terminal users. When *man* is piped to *lp*, however, it is undesirable for this squeezing to happen.

EXAMPLES

The –**p** allows arbitrary commands to be executed at the start of each file. Examples are:

more –p G *file1 file2*
: Examine each file starting with its last screenful.

more –p 100 *file1 file2*
: Examine each file starting with line 100 in the current position (usually the third line, so line 98 would be the first line written).

more –p /100 *file1 file2*
: Examine each file starting with the first line containing the string 100 in the current position

FUTURE DIRECTIONS

None.

SEE ALSO

pg.

CHANGE HISTORY

First released in Issue 4.

NAME

mv – move files

SYNOPSIS

```
mv [-fi] source_file target_file
mv [-fi] source_file... target_file
```

DESCRIPTION

In the first synopsis form, the *mv* utility moves the file named by the *source_file* operand to the *destination* specified by the *target_file*. This first synopsis form is assumed when the final operand does not name an existing directory.

In the second synopsis form, *mv* moves each file named by a *source_file* operand to a *destination* file in the existing directory named by the *target_dir* operand. The *destination* path for each *source_file* is the concatenation of the target directory, a single slash character, and the last pathname component of the *source_file*. This second form is assumed when the final operand names an existing directory.

If any operand specifies an existing file of a type not specified by the **XSH** specification, the behaviour is implementation-dependent.

For each *source_file* the following steps are taken:

1. If the destination path exists, the **–f** option is not specified, and either of the following conditions is true:

   a. The permissions of the destination path do not permit writing and the standard input is a terminal.

   b. The **–i** option is specified.

   The *mv* utility will write a prompt to standard error and read a line from standard input. If the response is not affirmative, *mv* will do nothing more with the current *source_file* and go on to any remaining *source_file*s.

2. The *mv* utility will perform actions equivalent to the **XSH** specification *rename*( ) function, called with the following arguments:

   a. The *source_file* operand is used as the *old* argument.

   b. The destination path is used as the *new* argument.

   If this succeeds, *mv* will do nothing more with the current *source_file* and go on to any remaining *source_file*s. If this fails for any reasons other than those described for the *errno* [EXDEV] in the **XSH** specification, *mv* will write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

3. If the destination path exists, and it is a file of type directory and *source_file* is not a file of type directory, or it is a file not of type directory and *source_file* is a file of type directory, *mv* will write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

4. If the destination path exists, *mv* will attempt to remove it. If this fails for any reason, *mv* will write a diagnostic message to standard error, do nothing more with the current *source_file*, and go on to any remaining *source_file*s.

5. The file hierarchy rooted in *source_file* will be duplicated as a file hierarchy rooted in the destination path. The following characteristics of each file in the file hierarchy will be duplicated:

   - the time of last data modification and time of last access
   - the user ID and group ID
   - the file mode.

   If the user ID, group ID or file mode of a regular file cannot be duplicated, the file mode bits S_ISUID and S_ISGID will not be duplicated.

   When files are duplicated to another file system, the implementation may require that the process invoking *mv* has read access to each file being duplicated.

   If the duplication of the file hierarchy fails for any reason, *mv* will write a diagnostic message to standard error, do nothing more with the current *source_file,* and go on to any remaining *source_file*s.

   If the duplication of the file characteristics fails for any reason, *mv* will write a diagnostic message to standard error, but this failure will not cause *mv* to modify its exit status.

6. The file hierarchy rooted in *source_file* will be removed. If this fails for any reason, *mv* will write a diagnostic message to the standard error, do nothing more with the current *source_file,* and go on to any remaining *source_file*s.

OPTIONS

The *mv* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–f Do not prompt for confirmation if the destination path exists. Any previous occurrences of the **–i** option will be ignored.

–i Prompt for confirmation if the destination path exists. Any previous occurrences of the **–f** option will be ignored.

Specifying more than one of the **–f** or **–i** options is not considered an error. The last option specified will determine the behaviour of *mv*.

OPERANDS

The following operands are supported:

source_file
A pathname of a file or directory to be moved.

target_file
A new pathname for the file or directory being moved.

target_dir
A pathname of an existing directory into which to move the input files.

STDIN

Used to read an input line in response to each prompt specified in **STDERR**. Otherwise, the standard input will not be used.

INPUT FILES

The input files specified by each *source_file* operand can be of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *mv*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the behaviour of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES
Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Prompts will be written to the standard error under the conditions specified in **DESCRIPTION**. The prompts will contain the *destination* pathname, but their format is otherwise unspecified. Otherwise, the standard error will be used only for diagnostic messages.

OUTPUT FILES
The output files may be of any file type.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 All input files were moved successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
If the copying or removal of *source_file* is prematurely terminated by a signal or error, *mv* may leave a partial copy of *source_file* at the source or destination. The *mv* utility will not modify both *source_file* and the destination path simultaneously; termination at any point will leave either *source_file* or the destination path complete.

APPLICATION USAGE
None.

EXAMPLES

If the current directory contains only files **a** (of any type defined by the **XSH** specification), **b** (also of any type), and a directory **c**:

```
mv a b c
mv c d
```

will result with the original files **a** and **b** residing in the directory **d** in the current directory.

FUTURE DIRECTIONS

None.

SEE ALSO

cp, *ln*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

newgrp – change to a new group

SYNOPSIS

```
newgrp [-l][group
```

OB

```
newgrp [-][group]
```

DESCRIPTION

The *newgrp* utility creates a new shell execution environment with a new real and effective group identification. Of the attributes listed in Section 2.12 on page 63, the new shell execution environment will retain the working directory, file creation mask and exported variables from the previous environment (that is, open files, traps, unexported variables, alias definitions, shell functions and *set* options may be lost). All other aspects of the process environment that are preserved by the *exec* family of functions in the **XSH** specification also are preserved by *newgrp*; whether other aspects are preserved is unspecified.

A failure to assign the new group identifications (for example, for security or password-related reasons) does not prevent the new shell execution environment from being created.

FIPS The *newgrp* utility affects the supplemental groups for the process as follows:

- On systems where the effective group ID is normally in the supplementary group list (or whenever the old effective group ID actually is in the supplementary group list):
  - — If the new effective group ID is also in the supplementary group list, *newgrp* will change the effective group ID.
  - — If the new effective group ID is not in the supplementary group list, *newgrp* will add the new effective group ID to the list, if there is room to add it.
- On systems where the effective group ID is not normally in the supplementary group list (or whenever the old effective group ID is not in the supplementary group list):
  - — If the new effective group ID is in the supplementary group list, *newgrp* will delete it.
  - — If the old effective group ID is not in the supplementary list, *newgrp* will add it if there is room.

Note: The **XSH** specification does not specify whether the effective group ID of a process is included in its supplementary group list.

With no operands, *newgrp* will change the effective group back to the groups identified in the
FIPS user's user entry, and will set the list of supplementary groups to that set in the user's group database entries.

If a password is required for the specified group, and the user is not listed as a member of that group in the group database, the user will be prompted to enter the correct password for that group. If the user is listed as a member of that group, no password will be requested. If no password is required for the specified group, it is implementation-dependent whether users not listed as members of that group can change to that group. Whether or not a password is required, implementation-dependent system accounting or security mechanisms may impose additional authorisation restrictions that may cause *newgrp* to write a diagnostic message and suppress the changing of the group identification.

OPTIONS

The *newgrp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**,
OB except that the obsolescent version uses – in a non-standard manner.

The following option is supported:

–l (The letter ell.)

OB **–** Change the environment to what would be expected if the user actually logged in again.

OPERANDS

The following operand is supported:

group A group name from the group database or a non-negative numeric group ID. Specifies the group ID to which the real and effective group IDs will be set. If *group* is a non-negative numeric string and exists in the group database as a group name (see *getgrnam*( )), the numeric group ID associated with that group name will be used as the group ID.

STDIN

Not used.

INPUT FILES

The file **/dev/tty** is used to read a single line of text for password checking, when one is required.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *newgrp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used for diagnostic messages and a prompt string for a password, if one is required. Diagnostic messages may be written in cases where the exit status is not available; see **EXIT STATUS**.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If *newgrp* succeeds in creating a new shell execution environment, whether or not the group identification was changed successfully, the exit status will be the exit status of the shell. Otherwise, the following exit value is returned:

>0 An error occurred.

CONSEQUENCES OF ERRORS

The invoking shell may terminate.

APPLICATION USAGE

There is no convenient way to enter a password into the Group Database. Use of group passwords is not encouraged, because by their very nature they encourage poor security practices. Group passwords may disappear in the future.

A common implementation of *newgrp* is that the current shell uses *exec* to overlay itself with *newgrp*, which in turn overlays itself with a new shell after changing group. On some systems, however, this may not occur and *newgrp* may be invoked as a subprocess.

The *newgrp* command is intended only for use from an interactive terminal. It does not offer a useful interface for the support of applications.

The exit status of *newgrp* is generally inapplicable. If *newgrp* is used in a script, in most cases it will successfully invoke a new shell and the rest of the original shell script will be bypassed when the new shell exits. Used interactively, *newgrp* displays diagnostic messages to indicate problems. But usage such as:

```
newgrp foo
echo $?
```

is not useful because the new shell might not have access to any status *newgrp* may have generated (and most historical systems do not provide this status). A zero status echoed here does not necessarily indicate that the user has changed to the new group successfully. Following *newgrp* with the *id* command provides a portable means of determining whether the group change was successful or not.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the **XSH** specification description of *exec*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

The *newgrp* utility is now mandatory; it is optional in Issue 3.

NAME

nice – invoke a utility with an altered system scheduling priority

SYNOPSIS

```
nice [-n increment] utility [argument...]
```

OB

```
nice [-increment] utility [argument...]
```

DESCRIPTION

The *nice* utility invokes a utility, requesting that it be run with a different system scheduling priority (see the definition of **system scheduling priority** in the **XBD** specification, **Chapter 2**, **Glossary**). With no options and only if the user has appropriate privileges, the executed utility is run with a system scheduling priority that is some implementation-dependent quantity less than or equal to the system scheduling priority of the current process. If the user lacks appropriate privileges to affect the system scheduling priority in the requested manner, the *nice* utility will not affect the system scheduling priority; in this case, a warning message may be written to standard error, but this will not prevent the invocation of *utility* or affect the exit status.

OPTIONS

OB The *nice* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** except that the obsolescent version allows a multi-digit decimal integer as an option name.

The following option is supported:

–n *increment*

OB *–increment*

Specify how the system scheduling priority of the executed utility will be adjusted. The *increment* option-argument is a positive or negative decimal integer that will be used to modify the system scheduling priority of the executed utility in an implementation-dependent manner.

Positive *increment* values cause a lower or unchanged system scheduling priority. Negative *increment* values may require appropriate privileges and will cause a higher or unchanged system scheduling priority.

The system scheduling priority is bounded in an implementation-dependent manner. If the requested *increment* would raise or lower the system scheduling priority of the executed utility beyond implementation-dependent limits, then the limit whose value was exceeded is used.

OPERANDS

The following operands are supported:

utility The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

argument

Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *nice*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path used to locate the utility to be invoked. See the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

If the *utility* utility is invoked, the exit status of *nice* will be the exit status of *utility;* otherwise, the *nice* utility will exit with one of the following values:

1-125 An error occurred in the *nice* utility.

126 The utility specified by *utility* was found but could not be invoked.

127 The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Note that, in the obsolescent version, **–5** is a positive *increment*, while **–-5** is a negative *increment*.

The only guaranteed portable uses of this utility are:

nice utility
Run *utility* with the default lower system scheduling priority.

nice **–n** <*positive integer*> *utility*
Run *utility* with a lower system scheduling priority.

On some systems they will have no discernible effect on the invoked utility and on some others they will be exactly equivalent.

Historical systems have frequently supported the <*positive integer*> up to 20. Since there is no error penalty associated with guessing a number that is too high, users without access to the system conformance document (to see what limits are actually in place) could use the historical 1 to 20 range or attempt to use very large numbers if the job should be truly low priority.

The system scheduling priority value of a process can be displayed using the command:

```
ps -o nice
```

The *command*, *env*, *nice*, *nohup*, *time* and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ''failure to find a utility'' from ''invoked utility exited with an error indication''. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ''normal error conditions'' and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

renice.

CHANGE HISTORY

First released in Issue 4.

NAME

nl – line numbering filter

SYNOPSIS

EX `nl [-p][-b type][-d delim][-h type][-i incr][-l num][-n format] [-s sep][-v startnum][-w width][file]`

DESCRIPTION

The *nl* utility reads lines from the named *file* or the standard input if no *file* is named and reproduces the lines to standard output. Lines are numbered on the left. Additional functionality may be provided in accordance with the command options in effect.

The *nl* utility 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 (for example, no numbering of header and footer lines while numbering blank lines only in the body).

The starts of logical page sections are signalled by input lines containing nothing but the following delimiter characters:

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

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

OPTIONS

OB The *nl* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the options can be intermingled with the optional *file* operand. Only one file can be named.

The following options are supported:

–b *type* Specify which logical page body lines are to be numbered. Recognised *types* and their meaning are:

- **a** Number all lines.
- **t** Number only lines with text consisting entirely of characters in the current locale's **graph** character classification.
- **n** No line numbering.
- **p***string* Number only lines that contain the basic regular expression specified in *string*.

The default *type* for logical page body is t (text lines numbered).

–h *type* Specify the same as **b** *type* except for header. The default *type* for logical page header is n (no lines numbered).

–f *type* Specify the same as **b** *type* except for footer. The default for logical page footer is n (no lines numbered).

–p Specify that numbering should not be restarted at logical page delimiters.

–v *startnum*
Specify the initial value used to number logical page lines. The default is 1.

–**i** *incr* Specify the increment value used to number logical page lines. The default is 1.

–**s** *sep* Specify the characters used in separating the line number and the corresponding text line. The default *sep* is a tab.

–**w** *width*
Specify the number of characters to be used for the line number. The default *width* is 6.

–**n** *format*
Specify the line numbering format. Recognised values are: **ln**, left justified, leading zeros suppressed; **rn**, right justified, leading zeros suppressed; **rz**, right justified, leading zeros kept. The default *format* is **rn** (right justified).

–**l** *num* Specify the number of blank lines to be considered as one. For example, –**l 2** results in only the second adjacent blank line being numbered (if the appropriate –**h a**, –**b a** or –**f a** option is set). The default is 1.

–**d** *delim*
Specify the delimiter characters that indicate the start of a logical page section. These can be changed from the default characters \: to two user-specified characters. If only one character is entered, the second character remains the default character :.

OPERANDS

The following operand is supported:

file A pathname of a text file to be line-numbered.

STDIN

The standard input is a text file that is used if no *file* operand is given.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *nl*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the behaviour of character classes within regular expressions, and for deciding which characters are in character class **graph** (for the –**b t**, –**f t** and –**h t** options).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output is a text file in the following format:

```
"%s%s%s", <line number>, <separator>, <input line>
```

where <*line number*> is one of the following numeric formats:

%6d when the **rn** format is used (the default; see **–n**)

%06d when the **rz** format is used

%–6d when the *ln* format is used

<empty>
when line numbers are suppressed for a portion of the page; the <*separator*> is also suppressed.

In the preceding list, the number 6 is the default width; the **–w** option can change this value.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
In using the **–d** *delim* option, care should be taken to escape characters that have special meaning to the command interpreter.

EXAMPLES
The command:

```
nl  –v 10  –i 10  –d \!+  file1
```

will number *file1* starting at line number 10 with an increment of 10. The logical page delimiter is !+. Note that the ! has to be escaped when using *csh* as a command interpreter because of its history substitution syntax. For *ksh* and *sh* the escape is not necessary, but will not do any harm.

FUTURE DIRECTIONS
The intermingling of the *file* operand with the options is an obsolescent feature that will be removed from a future issue.

SEE ALSO
pr.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guideline support mandated.

Internationalised environment variable support mandated.

NAME

nm – write the name list of an object file (**DEVELOPMENT**)

SYNOPSIS

EX `nm [-APv][-efox][ -g| -u][-t` *format*`] file...`

DESCRIPTION

The *nm* utility displays symbolic information appearing in the object file, executable file or object-file library named by *file*. If no symbolic information is available for a valid input file, the *nm* utility will report that fact, but not consider it an error condition.

EX The default base used when numeric values are written is decimal.

OPTIONS

The *nm* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–A Write the full pathname or library name of an object on each line.

EX **–e** Write only external (global) and static symbol information.

EX **–f** Produce full output. Write redundant symbols (*.text*, *.data* and *.bss*), normally suppressed.

–g Write only external (global) symbol information.

EX **–o** Write numeric values in octal (equivalent to **–t o**).

–P Write information in a portable output format, as specified in **STDOUT**.

–t *format*

Write each numeric value in the specified format. The format is dependent on the single character used as the *format* option-argument:

EX **d** The offset is written in decimal (default).

o The offset is written in octal.

x The offset is written in hexadecimal.

–u Write only undefined symbols.

–v Sort output by value instead of alphabetically.

EX **–x** Write numeric values in hexadecimal (equivalent to **–t x**).

OPERANDS

The following operand is supported:

file A pathname of an object file, executable file or object-file library.

STDIN

See **INPUT FILES**.

INPUT FILES

The input file must be an object file, an object-file library whose format is the same as those produced by the *ar* utility for link editing, or an executable file. The *nm* utility may accept additional implementation-dependent object library formats for the input file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *nm*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for character collation information for the symbol-name and symbol-value collation sequences.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If symbolic information is present in the input files, then for each file or for each member of an archive, the *nm* utility will write the following information to standard output. By default, the format is unspecified, but the output is sorted alphabetically by symbol name.

- Library or object name, if –**A** is specified.
- Symbol name.
- Symbol type, which is either one of the following single characters or an implementation-dependent type represented by a single character:

  A Global absolute symbol.

  a Local absolute symbol.

  B Global ‘‘bss’’ (that is, uninitialised data space) symbol.

  b Local bss symbol.

  D Global data symbol.

  d Local data symbol.

  T Global text symbol.

  t Local text symbol.

  U Undefined symbol.
- Value of the symbol.

- The size associated with the symbol, if applicable.

This information may be supplemented by additional information specific to the implementation.

If the **–P** option is specified, the previous information is displayed using the following portable format. The three versions differ depending on whether **–t d**, **–t o** or **–t x** was specified, respectively:

```
"%s%s %s %d %d\n", <library/object name>, <name>, <type>,
<value>, <size>

"%s%s %s %o %o\n", <library/object name>, <name>, <type>,
<value>, <size>

"%s%s %s %x %x\n", <library/object name>, <name>, <type>,
<value>, <size>
```

where <*library/object name*> is formatted as follows:

- If **–A** is not specified, <*library/object name*> is an empty string.
- If **–A** is specified and the corresponding *file* operand does not name a library:

  ```
  "%s: ", <file>
  ```

- If **–A** is specified and the corresponding *file* operand names a library. In this case, <*object file*> names the object file in the library containing the symbol being described:

  ```
  "%s[%s]: ", <file>, <object file>
  ```

If **–A** is not specified, then if more than one *file* operand is specified or if only one *file* operand is specified and it names a library, *nm* will write a line identifying the object containing the following symbols before the lines containing those symbols, in the form:

- If the corresponding *file* operand does not name a library:

  ```
  "%s:\n", <file>
  ```

- If the corresponding *file* operand names a library; in this case, <*object file*> is the name of the file in the library containing the following symbols:

  ```
  "%s[%s]:\n", <file>, <object file>
  ```

If **–P** is specified, but **–t** is not, the format is as if **–t x** had been specified.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Mechanisms for dynamic linking make this utility less meaningful when applied to an executable file because a dynamically linked executable may omit numerous library routines that would be found in a statically linked executable.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *c89*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

nohup – invoke a utility immune to hangups

SYNOPSIS

nohup *utility* **[***argument...***]**

DESCRIPTION

The *nohup* utility will invoke the utility named by the *utility* operand with arguments supplied as the *argument* operands. At the time the named *utility* is invoked, the SIGHUP signal is set to be ignored.

If the standard output is a terminal, all output written by the named *utility* to its standard output will be appended to the end of the file **nohup.out** in the current directory. If **nohup.out** cannot be created or opened for appending, the output will be appended to the end of the file **nohup.out** in the directory specified by the *HOME* environment variable. If neither file can be created or opened for appending, *utility* will not be invoked. If a file is created, the file's permission bits will be set to S_IRUSR | S_IWUSR.

If the standard error is a terminal, all output written by the named *utility* to its standard error will be redirected to the same file descriptor as the standard output.

OPTIONS

None.

OPERANDS

The following operands are supported:

utility The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

argument

Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *nohup*:

HOME Determine the pathname of the user's home directory: if the output file **nohup.out** cannot be created in the current directory, the *nohup* utility will use the directory named by *HOME* to create the file.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path that will be used to locate the utility to be invoked. See the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS
The *nohup* utility will take the standard action for all signals except that SIGHUP will be ignored.

STDOUT
If the standard output is not a terminal, the standard output of *nohup* will be the standard output generated by the execution of the *utility* specified by the operands. Otherwise, nothing will be written to the standard output.

STDERR
If the standard output is a terminal, a message will be written to the standard error, indicating the name of the file to which the output is being appended. The name of the file will be either **nohup.out** or **$HOME/nohup.out**.

OUTPUT FILES
If the standard output is a terminal, all output written by the named *utility* to the standard output and standard error is appended to the file **nohup.out**, which is created if it does not already exist.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

126 The utility specified by *utility* was found but could not be invoked.
127 An error occurred in the *nohup* utility or the utility specified by *utility* could not be found.

Otherwise, the exit status of *nohup* will be that of the utility specified by the *utility* operand.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The *command*, *env*, *nice*, *nohup*, *time* and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication’’. The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

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

Alternatively, the following command can be used to apply *nohup* to a complex command:

```
nohup sh -c 'complex-command-line'
```

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the **XSH** specification description of *signal*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

od – dump files in various formats

SYNOPSIS

```
od [-v][-A address_base][-j skip][-N count][-t type_string]...
[file...]
```

EX
```
od [-bcdosx][file] [[+]offset[.][b]]
```

DESCRIPTION

The *od* utility will write the contents of its input files to standard output in a user-specified format.

OPTIONS

The *od* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except
EX that the order of presentation of the **–t** options and the **–bcdosx** options is significant.

The following options are supported:

–A *address_base*
: Specify the input offset base (see **EXTENDED DESCRIPTION**). The *address_base* option-argument must be a character. The characters d, o and x specify that the offset base will be written in decimal, octal or hexadecimal, respectively. The character n specifies that the offset will not be written.

EX **–b** Interpret bytes in octal. This is equivalent to **–t o1**.

EX **–c** Interpret bytes as characters specified by the current setting of the LC_CTYPE category. 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. This is equivalent to **–t c**.

EX **–d** Interpret *word*s (two-byte units) in unsigned decimal. This is equivalent to **–t u2**.

–j *skip* Jump over *skip* bytes from the beginning of the input. The *od* utility will read or seek past the first *skip* bytes in the concatenated input files. If the combined input is not at least *skip* bytes long, the *od* utility will write a diagnostic message to standard error and exit with a non-zero exit status.

By default, the *skip* option-argument is interpreted as a decimal number. With a leading 0x or 0X, the offset is interpreted as a hexadecimal number; otherwise, with a leading 0, the offset will be interpreted as an octal number. Appending the character b, k or m to offset will cause it to be interpreted as a multiple of 512, 1024 or 1 048 576
EX bytes, respectively. If the *skip* number is hexadecimal, any appended b is considered to be the final hexadecimal digit.

–N *count*
: Format no more than *count* bytes of input. By default, *count* is interpreted as a decimal number. With a leading 0x or 0X, *count* is interpreted as a hexadecimal number; otherwise, with a leading 0, it is interpreted as an octal number. If *count* bytes of input (after successfully skipping, if **–j** *skip* is specified) are not available, it will not be considered an error; the *od* utility will format the input that is available.

EX **–o** Interpret *word*s (two-byte units) in octal. This is equivalent to **–t o2**.

EX PI **–s** Interpret *word*s (two-byte units) in signed decimal. This is equivalent to **–t d2**.

–t *type_string*

Specify one or more output types (see **EXTENDED DESCRIPTION**). The *type_string* option-argument must be a string specifying the types to be used when writing the input data. The string must consist of the type specification characters a, c, d, f, o, u and x, specifying named character, character, signed decimal, floating point, octal, unsigned decimal and hexadecimal, respectively. The type specification characters d, f, o, u and x can be followed by an optional unsigned decimal integer that specifies the number of bytes to be transformed by each instance of the output type. The type specification character f can be followed by an optional F, D or L indicating that the conversion should be applied to an item of type *float*, *double* or *long double*, respectively. The type specification characters d, o, u and x can be followed by an optional C, S, I or L indicating that the conversion should be applied to an item of type **char**, **short**, **int** or **long**, respectively. Multiple types can be concatenated within the same *type_string* and multiple **–t** options can be specified. Output lines are written for each type specified in the order in which the type specification characters are specified.

–v Write all input data. Without the **–v** option, any number of groups of output lines, which would be identical to the immediately preceding group of output lines (except for the byte offsets), will be replaced with a line containing only an asterisk (*).

EX **–x** Interpret *word*s (two-byte units) in hexadecimal. This is equivalent to **–t x2**.

EX Multiple types can be specified by using multiple **–bcdostx** options. Output lines are written for each type specified in the order in which the types are specified.

OPERANDS

The following operands are supported:

file A pathname of a file to be written. If no file operands are specified, the standard input will be used. If the first character of *file* is a plus sign (+) or the first character of the first file operand is numeric, no more than two operands are given, and none of the **–A**, **–j**, EX **–N** or **–t** options is specified, the operand is assumed to be an *offset*.

EX **[+]***offset***[.][b]**

The *offset* operand specifies the offset in the file where dumping is to commence. This operand 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, and none of the **–A**, **–j**, **–N** or **–t** options is specified, the offset argument must be preceded by +.

STDIN

The standard input is used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files can be any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *od*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_NUMERIC
Determine the locale for selecting the radix character used when writing floating-point formatted output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
See **EXTENDED DESCRIPTION**.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
The *od* utility copies sequentially each input file to standard output, transforming the input data
EX according to the output types specified by the **–t** options or the **–bcdosx** options. If no output type is specified, the default output is as if **–t o2** had been specified.

The number of bytes transformed by the output type specifier c may be variable depending on the LC_CTYPE category.

The default number of bytes transformed by output type specifiers d, f, o, u and x will correspond to the various C-language types as follows. If the *c89* compiler is present on the system, these specifiers will correspond to the sizes used by default in that compiler. Otherwise, these sizes are implementation-dependent.

- For the type specifier characters d, o, u and x, the default number of bytes will correspond to the size of the underlying implementation's basic integral data type. For these specifier characters, the implementation will support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **char**, **short**, **int** and **long.** These numbers can also be specified by an application as the characters C, S, I and L, respectively. The byte order used when interpreting numeric values is implementation-dependent, but will correspond to the order in which a constant of the corresponding type is stored in memory on the system.
- For the type specifier character f, the default number of bytes will correspond to the number of bytes in the underlying implementation's basic double precision floating-point data type. The implementation will support values of the optional number of bytes to be converted corresponding to the number of bytes in the C-language types **float, double** and **long double**. These numbers can also be specified by an application as the characters F, D and L, respectively.

The type specifier character a specifies that bytes are interpreted as named characters from the International Reference Version (IRV) of the ISO/IEC 646:1991 standard. Only the least

significant seven bits of each byte will be used for this type specification. Bytes with the values listed in the following table will be written using the corresponding names for those characters.

Value	Name	Value	Name	Value	Name	Value	Name
\000	**nul**	\001	**soh**	\002	**stx**	\003	**etx**
\004	**eot**	\005	**enq**	\006	**ack**	\007	**bel**
\010	**bs**	\011	**ht**	\012	**lf** or **nl***	\013	**vt**
\014	**ff**	\015	**cr**	\016	**so**	\017	**si**
\020	**dle**	\021	**dc1**	\022	**dc2**	\023	**dc3**
\024	**dc4**	\025	**nak**	\026	**syn**	\027	**etb**
\030	**can**	\031	**em**	\032	**sub**	\033	**esc**
\034	**fs**	\035	**gs**	\036	**rs**	\037	**us**
\040	**sp**	\177	**del**				

Table 3-10 Named Characters in *od*

Note: The **\012** value may be written either as **lf** or **nl**.

The type specifier character c specifies that bytes will be interpreted as characters specified by the current setting of the LC_CTYPE locale category. Characters listed in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v) will be written as the corresponding escape sequences, except that backslash will be written as a single backslash and a NUL will be written as \0. Other non-printable characters will be written as one three-digit octal number for each byte in the character. If the size of a byte on the system is greater than nine bits, the format used for non-printable characters is implementation-dependent. Printable multi-byte characters will be written in the area corresponding to the first byte of the character; the two-character sequence ** will be written in the area corresponding to each remaining byte in the character, as an indication that the character is continued. When either the **–j** *skip* or **–N** *count* option is specified along with the c type specifier, and this results in an attempt to start or finish in the middle of a multi-byte character, the result is implementation-dependent.

The input data is manipulated in blocks, where a block is defined as a multiple of the least common multiple of the number of bytes transformed by the specified output types. If the least common multiple is greater than 16, the results are unspecified. Each input block will be written as transformed by each output type, one per written line, in the order that the output types were specified. If the input block size is larger than the number of bytes transformed by the output type, the output type will sequentially transform the parts of the input block, and the output from each of the transformations will be separated by one or more blank characters.

If, as a result of the specification of the **–N** option or end-of-file being reached on the last input file, input data only partially satisfies an output type, the input will be extended sufficiently with null bytes to write the last byte of the input.

Unless **–A n** is specified, the first output line produced for each input block will be preceded by the input offset, cumulative across input files, of the next byte to be written. The format of the input offset is unspecified; however, it will not contain any blank characters, will start at the first character of the output line, and will be followed by one or more blank characters. In addition, the offset of the byte following the last byte written will be written after all the input data has been processed, but will not be followed by any blank characters.

If no **–A** option is specified, the input offset base is unspecified.

EXIT STATUS

The following exit values are returned:

0 All input files were processed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Applications are warned not to use filenames starting with + or a first operand starting with a numeric character so that the old functionality can be maintained by implementations, unless they specify one of the new options specified by the ISO/IEC 9945-2:1993 standard. To guarantee that one of these filenames will always be interpreted as a filename, an application could always specify the address base format with the **–A** option.

EXAMPLES

If a file containing 128 bytes with decimal values zero to 127, in increasing order, is supplied as standard input to the command:

```
od -A d -t a
```

on an implementation using an input block size of 16 bytes, the standard output, independent of the current locale setting, would be similar to:

```
0000000 nul soh stx etx eot enq ack bel  bs  ht  nl  vt  ff  cr  so  si
0000016 dle dc1 dc2 dc3 dc4 nak syn etb can  em sub esc  fs  gs  rs  us
0000032  sp   !   "   #   $   %   &   ’   (   )   *   +   ,   -   .   /
0000048   0   1   2   3   4   5   6   7   8   9   :   ;   <   =   >   ?
0000064   @   A   B   C   D   E   F   G   H   I   J   K   L   M   N   O
0000080   P   Q   R   S   T   U   V   W   X   Y   Z   [   \   ]   ^   _
0000096   ‘   a   b   c   d   e   f   g   h   i   j   k   l   m   n   o
0000112   p   q   r   s   t   u   v   w   x   y   z   {   |   }   ~ del
0000128
```

Note that this document allows **nl** or **lf** to be used as the name for the ISO/IEC 646:1991 standard IRV character with decimal value 10. The IRV names this character **lf** (line feed), but traditional implementations have referred to this character as newline (nl) and the POSIX locale character set symbolic name for the corresponding character is a newline character.

The command:

```
od -A o -t o2x2x -n 18
```

on a system with 32-bit words and an implementation using an input block size of 16 bytes could write 18 bytes in approximately the following format:

```
0000000 032056 031440 041123 042040 052516 044530 020043 031464
          342e   3320   4253   4420   554e   4958   2023   3334
             342e3320      42534420      554e4958      20233334
0000020 032472
          353a
             353a0000
0000022
```

The command:

```
od -A d -t f -t o4 -t x4 -n 24 -j 0x15
```

on a system with 64-bit doubles (for example, the IEEE Std 754 double precision floating-point format) would skip 21 bytes of input data and then write 24 bytes in approximately the following format:

```
0000000    1.00000000000000e+00    1.57350000000000e+01
         07774000000 00000000000 10013674121 35341217270
            3ff00000    00000000    402f3851    eb851eb8
0000016    1.40668230000000e+02
         10030312542 04370303230
            40619562    23e18698
0000024
```

FUTURE DIRECTIONS

All option and operand interfaces marked as extensions may be withdrawn from a future issue.

SEE ALSO

sed.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Issue 4, Version 2

The description of the **–c** option is made dependent on the current setting of the LC_CTYPE category, and a reference to the POSIX locale is deleted.

NAME

pack – compress files (**TO BE WITHDRAWN**)

SYNOPSIS

EX `pack [-f][-] file...`

DESCRIPTION

The *pack* utility attempts to store the specified files in a compressed form. Each input file is replaced by a packed file *file*.z. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file are preserved. If *pack* is successful, *file* will be removed. Packed files can be restored to their original form using *unpack* or *pcat*.

No packing will occur if:

- The file appears to be already packed.
- The filename has more than {NAME_MAX}–2 bytes.
- The file has links.
- The file is a directory.
- The file cannot be opened.
- The file is empty.
- No disk storage will be saved by packing.
- A file called *file***.z** already exists.
- The **.z** file cannot be created.
- An I/O error occurred during processing.

The last segment of the filename must contain no more than {NAME_MAX}–2 bytes to allow space for the appended **.z** extension.

OPTIONS

The *pack* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–f Force packing of files. This is useful for causing an entire directory to be packed even if some of the files will not benefit.

OPERANDS

The following operands are supported:

– Sets an internal flag that causes the number of times each byte is used, its relative frequency and the code for the byte to be written to standard output. Additional occurrences of – in place of *file* will cause the internal flag to be set and reset.

file A pathname of a file to be packed; *file* can include or omit the **.z** suffix.

STDIN

Not used.

INPUT FILES

The input files are regular files.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *pack*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If an error or signal occurs, the *file***.z** file is not created and the original file is unchanged.

STDOUT

The standard output is a text file containing one line for each file packed, with the following format used in the POSIX locale:

```
"pack: %s: %2.1f%% Compression\n", file, <percent compression>
```

or:

```
"pack: %s: no saving - file unchanged\n", file
```

If the – operand is specified and the internal flag is set, additional messages of unspecified format will be written to standard output as indicated in the **OPERANDS** section.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Packed files of unspecified format are created with names of the form *file***.z**.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The amount of compression obtained depends on the size of the input file and the character frequency distribution. Because a decoding tree may form the first part of each **.z** file, it is

usually not worthwhile to pack small files, 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. Object files, 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.

Packed files are not necessarily portable to other systems.

EXAMPLES

None.

FUTURE DIRECTIONS

This utility is being withdrawn from a future issue. The *compress* utility should be used instead; *compress* offers two advantages over *pack*:

- The algorithm used to create the output files is frequently more effective in reducing the sizes of files.
- The *compress* utility can compress data from its standard input, not just a named regular file. Thus, it is useful in pipelines.

SEE ALSO

pcat, *unpack*, *compress*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Split into a separate description.

Marked **TO BE WITHDRAWN**.

Utility Syntax Guideline support mandated.

Internationalised environment variable support made optional.

Issue 4, Version 2

The **DESCRIPTION** is clarified to state that the ownership, modes, access time, and modification time of the original file are preserved if the invoking process has appropriate privileges.

NAME

paste – merge corresponding or subsequent lines of files

SYNOPSIS

```
paste [-s][-d list] file...
```

DESCRIPTION

The *paste* utility will concatenate the corresponding lines of the given input files, and write the resulting lines to standard output.

The default operation of *paste* will concatenate the corresponding lines of the input files. The newline character of every line except the line from the last input file will be replaced with a tab character.

If an end-of-file condition is detected on one or more input files, but not all input files, *paste* will behave as though empty lines were read from the files on which end-of-file was detected, unless the **–s** option is specified.

OPTIONS

The *paste* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–d *list* Unless a backslash character appears in *list*, each character in *list* is an element specifying a delimiter character. If a backslash character appears in *list*, the backslash character and one or more characters following it are an element specifying a delimiter character as described below. These elements specify one or more delimiters to use, instead of the default tab character, to replace the newline character of the input lines. The elements in *list* are used circularly; that is, when the list is exhausted the first element from the list is reused. When the **–s** option is specified:

- The last newline character in a file will not be modified.
- The delimiter will be reset to the first element of list after each *file* operand is processed.

When the **–s** option is not specified:

- The newline characters in the file specified by the last *file* operand will not be modified.
- The delimiter will be reset to the first element of list each time a line is processed from each file.

If a backslash character appears in *list*, it and the character following it will be used to represent the following delimiter characters:

\n Newline character.

\t Tab character.

**** Backslash character.

\0 Empty string (not a null character). If \0 is immediately followed by the character x, the character X, or any character defined by the LC_CTYPE **digit** keyword (see the **XBD** specification, **Chapter 5**, **Locale**), the results are unspecified.

If any other characters follow the backslash, the results are unspecified.

–s Concatenate all of the lines of each separate input file in command line order. The newline character of every line except the last line in each input file will be replaced with the tab character, unless otherwise specified by the **–d** option.

OPERANDS

The following operand is supported:

file A pathname of an input file. If – is specified for one or more of the *file*s, the standard input will be used; the standard input will be read one line at a time, circularly, for each instance of –. Implementations support pasting of at least 12 *file* operands.

STDIN

The standard input will be used only if one or more *file* operands is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files, except that line lengths will be unlimited.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *paste*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Concatenated lines of input files will be separated by the tab character (or other characters under the control of the **–d** option) and terminated by a newline character.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If one or more input files cannot be opened when the **–s** option is not specified, a diagnostic message will be written to standard error, but no output will be written to standard output. If the **–s** option is specified, the *paste* utility will provide the default behaviour described in the **XBD** specification, **Section 2.1**, **Utility Description Defaults**.

APPLICATION USAGE

When the escape sequences of the *list* option-argument are used in a shell script, they must be quoted; otherwise, the shell treats the \ as a special character.

Portable applications should only use the specific backslash escaped delimiters presented in this document. Historical implementations treat \x, where x is not in this list, as x, but future implementations are free to expand this list to recognise other common escapes similar to those accepted by *printf* and other standard utilities.

Most of the standard utilities work on text files. The *cut* utility can be used to turn files with arbitrary line lengths into a set of text files containing the same data. The *paste* utility can be used to create (or recreate) files with arbitrary line lengths. For example, if **file** contains long lines:

```
cut -b 1-500 -n file > file1
cut -b 501- -n file > file2
```

creates **file1** (a text file) with lines no longer than 500 bytes (plus the newline character) and **file2** that contains the remainder of the data from **file**. Note that file2 will not be a text file if there are lines in file that are longer than 500 + {LINE_MAX} bytes. The original file can be recreated from **file1** and **file2** using the command:

```
paste -d "\0" file1 file2 > file
```

The commands:

```
paste -d "\0" ...
paste -d "" ...
```

are not necessarily equivalent; the latter is not specified by this document and may result in an error. The construct \0 is used to mean ‘‘no separator’’ because historical versions of *paste* did not follow the syntax guidelines, and the command:

```
paste -d"" ...
```

could not be handled properly by *getopt*( ).

EXAMPLES

1. Write out a directory in four columns:

   ```
   ls | paste - - - -
   ```

2. Combine pairs of lines from a file into single lines:

   ```
   paste -s -d "\t\n" file
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

cut, *grep*, *pr*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

patch – apply changes to files

SYNOPSIS

```
patch [-blNR][ -c| -e| -n][-d dir][-D define][-i patchfile]
[-o outfile][-p num][-r rejectfile][file]
```

DESCRIPTION

The *patch* utility reads a source (patch) file containing any of the three forms of difference (diff) listings produced by the *diff* utility (normal, context or in the style of *ed*) and apply those differences to a file. By default, *patch* reads from the standard input.

The *patch* utility attempts to determine the type of the *diff* listing, unless overruled by a **–c**, **–e** or **–n** option.

If the patch file contains more than one patch, *patch* will attempt to apply each of them as if they came from separate patch files. (In this case the name of the patch file must be determinable for each *diff* listing.)

OPTIONS

The *patch* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–b Save a copy of the original contents of each modified file, before the differences are applied, in a file of the same name with the suffix **.orig** appended to it. If the file already exists, it will be overwritten; if multiple patches are applied to the same file, the **.orig** file will be written only for the first patch. When the **–o** *outfile* option is also specified, *file***.orig** will not be created but, if *outfile* already exists, *outfile***.orig** will be created.

–c Interpret the patch file as a context difference (the output of the utility *diff* when the **–c** or **–C** options are specified).

–d *dir* Change the current directory to *dir* before processing as described in **EXTENDED DESCRIPTION**.

–D *define*

Mark changes with the C preprocessor construct:

```
#ifdef define
...
#endif
```

The option-argument *define* will be used as the differentiating symbol.

–e Interpret the patch file as an *ed* script, rather than a *diff* script.

–i *patchfile*

Read the patch information from the file named by the pathname *patchfile*, rather than the standard input.

–l (The letter ell.) Cause any sequence of blank characters in the difference script to match any sequence of blank characters in the input file. Other characters will be matched exactly.

–n Interpret the script as a normal difference.

–N Ignore patches where the differences have already been applied to the file; by default, already-applied patches are rejected.

–o *outfile*
Instead of modifying the files (specified by the *file* operand or the difference listings) directly, write a copy of the file referenced by each patch, with the appropriate differences applied, to *outfile*. Multiple patches for a single file will be applied to the intermediate versions of the file created by any previous patches, and will result in multiple, concatenated versions of the file being written to *outfile*.

–p *num*
For all pathnames in the patch file that indicate the names of files to be patched, delete *num* pathname components from the beginning of each pathname. If the pathname in the patch file is absolute, any leading slashes are considered the first component (that is, **–p 1** removes the leading slashes). Specifying **–p 0** causes the full pathname to be used. If **–p** is not specified, only the basename (the final pathname component) is used.

–R Reverse the sense of the patch script; that is, assume that the difference script was created from the new version to the old version. The **–R** option cannot be used with *ed* scripts. The *patch* utility attempts to reverse each portion of the script before applying it. Rejected differences will be saved in swapped format. If this option is not specified, and until a portion of the patch file is successfully applied, *patch* attempts to apply each portion in its reversed sense as well as in its normal sense. If the attempt is successful, the user will be prompted to determine if the **–R** option should be set.

–r *rejectfile*
Override the default reject filename. In the default case, the reject file will have the same name as the output file, with the suffix **.rej** appended to it. See **Patch Application** on page 551.

OPERANDS

The following operand is supported:

file A pathname of a file to patch.

STDIN

See **INPUT FILES**.

INPUT FILES

Input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *patch*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

LC_TIME
Determine the locale for recognising the format of file timestamps written by the *diff* utility in a context-difference input file.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used for diagnostic and informational messages.

OUTPUT FILES
The output of the *patch* utility, the save files (**.orig** suffixes) and the reject files (**.rej** suffixes) will be text files.

EXTENDED DESCRIPTION
A patchfile may contain patching instructions for more than one file; filenames are determined as specified in **Filename Determination** on page 551. When the **–b** option is specified, for each patched file, the original will be saved in a file of the same name with the suffix **.orig** appended to it.

For each patched file, a reject file may also be created as noted in **Patch Application** on page 551. In the absence of a **–r** option, the name of this file will be formed by appending the suffix **.rej** to the original filename.

Patchfile Format

The patch file must contain zero or more lines of header information followed by one or more patches. Each patch must contain zero or more lines of filename identification in the format produced by *diff* **–c**, and one or more sets of *diff* output, which are customarily called hunks.

The *patch* utility recognises the following expression in the header information:

Index: *pathname*
The file to be patched is named *pathname*.

If all lines (including headers) within a patch begin with the same leading sequence of blank characters, the *patch* utility will remove this sequence before proceeding. Within each patch, if the type of difference is context, the *patch* utility recognises the following expressions:

******* *filename timestamp*
The patches arose from *filename*.

– – – *filename timestamp*
The patches should be applied to *filename*.

Each hunk within a patch must be the *diff* output to change a line range within the original file. The line numbers for successive hunks within a patch must occur in ascending order.

Filename Determination

If no *file* operand is specified, *patch* performs the following steps to obtain a pathname:

1. If the patch contains the strings *** and −−−, the *patch* utility strips components from the beginning of each pathname (depending on the presence or value of the **−p** option), then tests for the existence of both files in the current directory (or directory specified with the **−d** option). If both files exist, *patch* assumes that no pathname can be obtained from this step.
2. If the header information contains a line with the string **Index:**, *patch* utility strips components from the beginning of the pathname (depending on **−p**), then tests for the existence of this file in the current directory (or directory specified with the **−d** option).

EX 3. If an **SCCS** directory exists in the current directory, *patch* will attempt to perform a *get* **−e** SCCS/s.*filename* command to retrieve an editable version of the file.

4. If no pathname can be obtained by applying the previous steps, or if the pathnames obtained do not exist, *patch* will write a prompt to standard output and request a filename interactively from standard input.

Patch Application

If the **−c**, **−e** or **−n** option is present, the *patch* utility will interpret information within each hunk as a context difference, an *ed* difference or a normal difference, respectively. In the absence of any of these options, the *patch* utility determines the type of difference based on the format of information within the hunk.

For each hunk, the *patch* utility begins to search for the place to apply the patch at the line number at the beginning of the hunk, plus or minus any offset used in applying the previous hunk. If lines matching the hunk context are not found, *patch* scans both forwards and backwards at least 1000 bytes for a set of lines that match the hunk context.

If no such place is found and it is a context difference, then another scan will take place, ignoring the first and last line of context. If that fails, the first two and last two lines of context will be ignored and another scan will be made. Implementations may search more extensively for installation locations.

If no location can be found, the *patch* utility will append the hunk to the reject file. The rejected hunk will be written in context-difference format regardless of the format of the patch file. If the input was a normal or *ed*-style difference, the reject file may contain differences with zero lines of context. The line numbers on the hunks in the reject file may be different from the line numbers in the patch file since they will reflect the approximate locations for the failed hunks in the new file rather than the old one.

If the type of patch is an *ed* diff, the implementation may accomplish the patching by invoking the *ed* utility.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
1 One or more lines were written to a reject file.
>1 An error occurred.

CONSEQUENCES OF ERRORS

Patches that cannot be correctly placed in the file will be written to a reject file.

APPLICATION USAGE

The **–R** option will not work with *ed* scripts because there is too little information to reconstruct the reverse operation.

The **–p** option makes it possible to customise a patchfile to local user directory structures without manually editing the patchfile. For example, if the filename in the patch file was:

```
/curds/whey/src/blurfl/blurfl.c
```

Setting **–p 0** gives the entire pathname unmodified; **–p 1** gives:

```
curds/whey/src/blurfl/blurfl.c
```

without the leading slash, **–p 4** gives:

```
blurfl/blurfl.c
```

and not specifying **–p** at all gives:

```
blurfl.c .
```

When using **–b** in some file system implementations, the saving of a **.orig** file may produce unwanted results. In the case of 12, 13 or 14-character filenames, on file systems supporting 14-character maximum filenames, the **.orig** file will overwrite the new file.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ed, *diff*.

CHANGE HISTORY

First released in Issue 4.

NAME

pathchk – check pathnames

SYNOPSIS

`pathchk [-p] pathname...`

DESCRIPTION

The *pathchk* utility will check that one or more pathnames are valid (that is, they could be used to access or create a file without causing syntax errors) and portable (that is, no filename truncation will result). More extensive portability checks are provided by the **–p** option.

By default, the *pathchk* utility will check each component of each *pathname* operand based on the underlying file system. A diagnostic will be written for each *pathname* operand that:

- is longer than {PATH_MAX} bytes (see **Pathname Variable Values** in the **XSH** specification **<limits.h>** description)
- contains any component longer than {NAME_MAX} bytes in its containing directory
- contains any component in a directory that is not searchable
- contains any character in any component that is not valid in its containing directory.

The format of the diagnostic message is not specified, but will indicate the error detected and the corresponding *pathname* operand.

It will not be considered an error if one or more components of a *pathname* operand do not exist as long as a file matching the pathname specified by the missing components could be created that does not violate any of the checks specified above.

OPTIONS

The *pathchk* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–p Instead of performing checks based on the underlying file system, write a diagnostic for each *pathname* operand that:

- is longer than {_POSIX_PATH_MAX} bytes (see **Minimum Values** in the **XSH** specification **<limits.h>** description)
- contains any component longer than {_POSIX_NAME_MAX} bytes
- contains any character in any component that is not in the portable filename character set.

OPERANDS

The following operand is supported:

pathname
A pathname to be checked.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *pathchk*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All *pathname* operands passed all of the checks.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *test* utility can be used to determine if a given pathname names an existing file; it will not, however, give any indication of whether or not any component of the pathname was truncated in a directory where the {_POSIX_NO_TRUNC} feature is not in effect. The *pathchk* utility does not check for file existence; it performs checks to determine if a pathname does exist or could be created with no pathname component truncation.

The *noclobber* option in the shell (see the *set* special built-in) can be used to atomically create a file. As with all file creation semantics in the **XSH** specification, it guarantees atomic creation, but still depends on applications to agree on conventions and cooperate on the use of files after they have been created.

EXAMPLES

To verify that all pathnames in an imported data interchange archive are legitimate and unambiguous on the current system:

```
pax -f archive | sed -e '/ == .*/s///' | xargs pathchk
if [ $? -eq 0 ]
then
       pax -r -f archive
else
       echo Investigate problems before importing files.
       exit 1
fi
```

To verify that all files in the current directory hierarchy could be moved to any system conforming to the **XSH** specification that also supports the *pax* utility:

```
find . -print | xargs pathchk -p
if [ $? -eq 0 ]
then
       pax -w -f archive .
else
       echo Portable archive cannot be created.
       exit 1
fi
```

To verify that a user-supplied pathname names a readable file and that the application can create a file extending the given path without truncation and without overwriting any existing file:

```
case $- in
       *C*)    reset="";;
       *)      reset="set +C"
              set -C;;
esac
test -r "$path" && pathchk "$path.out" &&
       rm "$path.out" > "$path.out"
if [ $? -ne 0 ]; then
       printf "%s: %s not found or %s.out fails \
creation checks.\n" $0 "$path" "$path"
       $reset # reset the noclobber option in case a trap
              # on EXIT depends on it
       exit 1
fi
$reset
PROCESSING < "$path" > "$path.out"
```

The following assumptions are made in this example:

1. **PROCESSING** represents the code that will be used by the application to use **$path** once it is verified that **$path.out** will work as intended.
2. The state of the *noclobber* option is unknown when this code is invoked and should be set on exit to the state it was in when this code was invoked. (The **reset** variable is used in this example to restore the initial state.)

3. Note the usage of:

   ```
   rm "$path.out" > "$path.out"
   ```

   a. The *pathchk* command has already verified, at this point, that **$path.out** will not be truncated.

   b. With the *noclobber* option set, the shell will verify that **$path.out** does not already exist before invoking *rm*.

   c. If the shell succeeded in creating **$path.out**, *rm* will remove it so that the application can create the file again in the **PROCESSING** step.

   d. If the **PROCESSING** step wants the file to exist already when it is invoked, the:

      ```
      rm "$path.out" > "$path.out"
      ```

      should be replaced with:

      ```
      > "$path.out"
      ```

      which will verify that the file did not already exist, but leave **$path.out** in place for use by **PROCESSING**.

FUTURE DIRECTIONS

None.

SEE ALSO

test, Section 2.7 on page 40.

CHANGE HISTORY

First released in Issue 4.

NAME

pax – portable archive interchange

SYNOPSIS

```
pax [-cdnv][-f archive][-s replstr]...[pattern...]

pax -r[-cdiknuv][-f archive][-o options]...[-p string]...
[-s replstr]...[pattern...]

pax -w[-dituvX][-b blocksize][[-a][-f archive][-o options]...
[-s replstr]...[-x format][file...]

pax -r -w[-diklntuvX][-p string]...[-s replstr]...[file...]
directory
```

DESCRIPTION

The *pax* utility reads, writes and writes lists of the members of archive files and copy directory hierarchies. A variety of archive formats are supported; see the –**x** *format* option.

The action to be taken depends on the presence of the –**r** and –**w** options. The four combinations of –**r** and –**w** are referred to as the four modes of operation: *list*, *read*, *write* and *copy* modes, corresponding respectively to the four forms shown in the **SYNOPSIS**.

list In list mode (when neither –**r** nor –**w** are specified), *pax* writes the names of the members of the archive file read from the standard input, with pathnames matching the specified patterns, to standard output. If a named file is of type directory, the file hierarchy rooted at that file will be written out as well.

read In read mode (when –**r** is specified, but –**w** is not), *pax* extracts the members of the archive file read from the standard input, with pathnames matching the specified patterns. If an extracted file is of type directory, the file hierarchy rooted at that file will be extracted as well. The extracted files is created relative to the current file hierarchy.

The ownership, access and modification times, and file mode of the restored files are discussed under the –**p** option.

write In write mode (when –**w** is specified, but –**r** is not), *pax* writes the contents of the file operands to the standard output in an archive format. If no *file* operands are specified, a list of files to copy, one per line, will be read from the standard input. A file of type directory will include all of the files in the file hierarchy rooted at the file.

copy In copy mode (when both –**r** and –**w** are specified), *pax* copies the file operands to the destination directory.

If no *file* operands are specified, a list of files to copy, one per line, will be read from the standard input. A file of type directory will include all of the files in the file hierarchy rooted at the file.

The effect of the copy is as if the copied files were written to an archive file and then subsequently extracted, except that there may be hard links between the original and the copied files. If the destination directory is a subdirectory of one of the files to be copied, the results are unspecified. If the destination directory is a file of a type not defined by the **XSH** specification, the results are implementation-dependent; otherwise it is an error for the file named by the directory operand not to exist, not be writable by the user, or not be a file of type directory.

In read or copy modes, if intermediate directories are necessary to extract an archive member, *pax* will perform actions equivalent to the **XSH** specification *mkdir*( ) function, called with the following arguments:

- the intermediate directory used as the *path* argument
- the value of the bitwise inclusive OR of S_IRWXU, S_IRWXG and S_IRWXO as the *mode* argument.

If any specified *pattern* or *file* operands are not matched by at least one file or archive member, *pax* will write a diagnostic message to standard error for each one that did not match and exit with a non-zero exit status.

The supported archive formats are automatically detected on input. The default output archive format is implementation-dependent.

A single archive can span multiple files. The *pax* utility determines, in an implementation-dependent manner, what file to read or write as the next file.

If the selected archive format supports the specification of linked files, it is an error if these files cannot be linked when the archive is extracted. Any of the various names in the archive that represent a file can be used to select the file for extraction.

OPTIONS

The *pax* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the order of presentation of the **–s** options is significant.

The following options are supported:

–r Read an archive file from standard input.

–w Write files to the standard output in the specified archive format.

–a Append files to the end of the archive. It is implementation-dependent which devices on the system support appending. Additional file formats unspecified by this document may impose restrictions on appending.

–b *blocksize*
Block the output at a positive decimal integer number of bytes per write to the archive file. Devices and archive formats may impose restrictions on blocking. Blocking is automatically determined on input. Portable applications must not specify a *blocksize* value larger than 32 256. Default blocking when creating archives depends on the archive format. (See the **–x** option below.)

–c Match all file or archive members except those specified by the *pattern* or *file* operands.

–d Cause files of type directory being copied or archived or archive members of type directory being extracted to match only the file or archive member itself and not the file hierarchy rooted at the file.

–f *archive*
Specify the pathname of the input or output archive, overriding the default standard input (in list or read modes) or standard output (write mode).

–i Interactively rename files or archive members. For each archive member matching a *pattern* operand or file matching a *file* operand, a prompt will be written to the file **/dev/tty**. The prompt will contain the name of the file or archive member, but the format is otherwise unspecified. A line will then be read from **/dev/tty**. If this line is blank, the file or archive member will be skipped. If this line consists of a single period, the file or archive member will be processed with no modification to its name.

Otherwise, its name will be replaced with the contents of the line. The *pax* utility will immediately exit with a non-zero exit status if end-of-file is encountered when reading a response or if **/dev/tty** cannot be opened for reading and writing.

–k Prevent the overwriting of existing files.

–l (The letter ell.) Link files. In copy mode, hard links will be made between the source and destination file hierarchies whenever possible.

–n Select the first archive member that matches each *pattern* operand. No more than one archive member will be matched for each pattern (although members of type directory will still match the file hierarchy rooted at that file).

–o *options*
Provide information to the implementation to modify the algorithm for extracting or writing files that is specific to the file format specified by **–x**. This issue does not specify any such options and a portable application must not use the **–o** option.

–p *string*
Specify one or more file characteristic options (privileges). The *string* option-argument must be a string specifying file characteristics to be retained or discarded on extraction. The string consists of the specification characters a, e, m, o and p. Other, implementation-dependent, characters can be included. Multiple characteristics can be concatenated within the same string and multiple **–p** options can be specified. The meaning of the specification characters are as follows:

a Do not preserve file access times.

e Preserve the user ID, group ID, file mode bits (see **file mode bits** in the **XBD** specification, **Chapter 2**, **Glossary**), access time, modification time and any other, implementation-dependent, file characteristics.

m Do not preserve file modification times.

o Preserve the user ID and group ID.

p Preserve the file mode bits. Other, implementation-dependent file-mode attributes may be preserved.

In the preceding list, ‘‘preserve’’ indicates that an attribute stored in the archive will be given to the extracted file, subject to the permissions of the invoking process; otherwise, the attribute will be determined as part of the normal file creation action.

If neither the e nor the o specification character is specified, or the user ID and group ID are not preserved for any reason, *pax* will not set the S_ISUID and S_ISGID bits of the file mode.

If the preservation of any of these items fails for any reason, *pax* will write a diagnostic message to standard error. Failure to preserve these items will affect the final exit status, but will not cause the extracted file to be deleted.

If file-characteristic letters in any of the *string* option-arguments are duplicated or conflict with each other, the ones given last will take precedence. For example, if **–p eme** is specified, file modification times will be preserved.

–**s** *replstr*

Modify file or archive member names named by *pattern* or *file* operands according to the substitution expression *replstr*, using the syntax of the *ed* utility. The concepts of ''address'' and ''line'' are meaningless in the context of the *pax* utility, and must not be supplied. The format is:

```
–s /old/new/[gp]
```

where as in *ed*, *old* is a basic regular expression and *new* can contain an ampersand, *n* (where *n* is a digit) backreferences, or subexpression matching. The *old* string also is permitted to contain newline characters.

Any non-null character can be used as a delimiter (/ shown here). Multiple –**s** expressions can be specified; the expressions will be applied in the order specified, terminating with the first successful substitution. The optional trailing g is as defined in the *ed* utility. The optional trailing p causes successful substitutions to be written to standard error. File or archive member names that substitute to the empty string are ignored when reading and writing archives.

–**t** Cause the access times of the archived files to be the same as they were before being read by *pax*.

–**u** Ignore files that are older (having a less recent file modification time) than a pre-existing file or archive member with the same name. In read mode, an archive member with the same name as a file in the file system will be extracted if the archive member is newer than the file. In write mode, an archive file member with the same name as a file in the file system will be superseded if the file is newer than the archive member. It is unspecified if this is accomplished by actual replacement in the archive or by appending to the archive. In copy mode, the file in the destination hierarchy will be replaced by the file in the source hierarchy or by a link to the file in the source hierarchy if the file in the source hierarchy is newer.

–**v** In list mode, produce a verbose table of contents (see **STDOUT**). Otherwise, write archive member pathnames to standard error (see **STDERR**).

–**x** *format*

Specify the output archive format. The *pax* utility recognises the following formats:

cpio The extended *cpio* interchange format; see **EXTENDED DESCRIPTION**. The default *blocksize* for this format for character special archive files is 5120. Implementations support all *blocksize* values less than or equal to 32 256 that are multiples of 512.

ustar The extended *tar* interchange format; see **EXTENDED DESCRIPTION**. The default *blocksize* for this format for character special archive files is 10 240. Implementations support all *blocksize* values less than or equal to 32 256 that are multiples of 512.

Implementation-dependent formats will specify a default block size as well as any other block sizes supported for character special archive files.

Any attempt to append to an archive file in a format different from the existing archive format will cause *pax* to exit immediately with a non-zero exit status.

–**X** When traversing the file hierarchy specified by a pathname, *pax* will not descend into directories that have a different device ID (**st_dev**, see the **XSH** specification *stat*( )).

The options that operate on the names of files or archive members (–**c**, –**i**, –**n**, –**s**, –**u** and –**v**) interact as follows. In read mode, the archive members are selected based on the user-specified

pattern operands as modified by the **–c**, **–n** and **–u** options. Then, any **–s** and **–i** options will modify, in that order, the names of the selected files. The **–v** option will write names resulting from these modifications.

In write mode, the files are selected based on the user-specified pathnames as modified by the **–n** and **–u** options. Then, any **–s** and **–i** options will, in that order, modify the names of these selected files. The **–v** option will write names resulting from these modifications.

If both the **–u** and **–n** options are specified, *pax* does not consider a file selected unless it is newer than the file to which it is compared.

OPERANDS

The following operands are supported:

directory
: The destination directory pathname for copy mode.

file A pathname of a file to be copied or archived.

pattern A pattern matching one or more pathnames of archive members. A pattern must be given in the name-generating notation of the pattern matching notation in Section 2.13 on page 64, including the filename expansion rules in Section 2.13.3 on page 66. The default, if no *pattern* is specified, is to select all members in the archive.

STDIN

In write mode, the standard input is used only if no *file* operands are specified. It must be a text file containing a list of pathnames, one per line, without leading or trailing blank characters.

In list and read modes, the standard input must be an archive file.

Otherwise, the standard input will not be used.

INPUT FILES

The input file named by the *archive* option-argument, or standard input when the archive is read from there, will be a file formatted according to one of the specifications in the **EXTENDED DESCRIPTION** or some other, implementation-dependent, format.

The file **/dev/tty** is used to write prompts and read responses.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *pax*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
: Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the pattern matching expressions for the *pattern* operand, the basic regular expression for the **–s** option, and the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and

input files), the behaviour of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category, and pattern matching.

LC_MESSAGES
Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format and contents of date and time strings when the –**v** option is specified.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
In write mode, if –**f** is not specified, the standard output will be the archive formatted according to one of the specifications in **EXTENDED DESCRIPTION**, or some other implementation-dependent format. (See –**x** *format*.)

In list mode, the table of contents of the selected archive members will be written to standard output using the following format:

"%s\n", <*pathname*>

If the –**v** option is specified in list mode, the table of contents of the selected archive members will be written to standard output using the following formats:

For pathnames representing hard links to previous members of the archive:

"%s∆==∆%s\n", <*ls –l listing*>, <*linkname*>

For all other pathnames:

"%s\n", <*ls –l listing*>

where <*ls –l listing*> is the format specified by the *ls* utility with the –**l** option. When writing pathnames in this format, it is unspecified what is written for fields for which the underlying archive format does not have the correct information, although the correct number of blank-character-separated fields will be written.

In list mode, standard output will not be buffered more than a line at a time.

STDERR
If –**v** is specified in read, write or copy modes, *pax* will write the pathnames it processes to the standard error output using the following format:

"%s\n", <*pathname*>

These pathnames will be written as soon as processing is begun on the file or archive member, and will be flushed to standard error. The trailing newline character, which will not be buffered, will be written when the file has been read or written.

If the –**s** option is specified, and the replacement string has a trailing p, substitutions will be written to standard error in the following format:

"%s∆>>∆%s\n", <*original pathname*>, <*new pathname*>

In all operating modes of *pax*, optional messages of unspecified format concerning the input archive format and volume number, the number of files, blocks, volumes and media parts as well as other diagnostic messages may be written to standard error.

In all formats, for both standard output and standard error, it is unspecified how non-printable characters in pathnames or linknames are written.

OUTPUT FILES

In read mode, the extracted or copied output files are of the archived file type.

In write mode, the output file named by the **–f** option-argument is a file formatted according to one of the specifications in **EXTENDED DESCRIPTION**, or some other, implementation-dependent, format.

EXTENDED DESCRIPTION

Extended cpio Format

The octet-oriented *cpio* archive format is a series of entries, each comprising a header that describes the file, the name of the file and then the contents of the file.

An archive may be recorded as a series of fixed-size blocks of octets. This blocking will be used only to make physical I/O more efficient. The last group of blocks always will be at the full size.

For the octet-oriented *cpio* archive format, the individual entry information will be in the order indicated and described by the following table:

Header Field Name	**Length (in octets)**	**Interpreted as**
c_magic	6	Octal number
c_dev	6	Octal number
c_ino	6	Octal number
c_mode	6	Octal number
c_uid	6	Octal number
c_gid	6	Octal number
c_nlink	6	Octal number
c_rdev	6	Octal number
c_mtime	11	Octal number
c_namesize	6	Octal number
c_filesize	11	Octal number
Filename Field Name	**Length**	**Interpreted as**
c_name	**c_namesize**	Pathname string
File Data Field Name	**Length**	**Interpreted as**
c_filedata	**c_filesize**	Data

Table 3-11 Octet-oriented *cpio* Archive Entry

The cpio Header

For each file in the archive, a header as defined previously will be written. The information in the header fields will be written as streams of the ISO/IEC 646:1991 standard characters interpreted as octal numbers. The octal numbers will be extended to the necessary length by appending the ISO/IEC 646:1991 standard IRV zeros at the most-significant-digit end of the number; the result is written to the most-significant-digit of the stream of octets first. The fields are interpreted as follows:

c_magic
Identify the archive as being a transportable archive by containing the identifying value "070707".

c_dev
c_ino Contains values that uniquely identify the file within the archive (that is, no files will contain the same pair of **c_dev** and **c_ino** values unless they are links to the same file). The values will be determined in an unspecified manner.

c_mode Contains the file type and access permissions as defined in the following table:

File Permissions		
Name	**Value**	**Indicates**
C_IRUSR	000 400	Read by owner.
C_IWUSR	000 200	Write by owner.
C_IXUSR	000 100	Execute by owner.
C_IRGRP	000 040	Read by group.
C_IWGRP	000 020	Write by group.
C_IXGRP	000 010	Execute by group.
C_IROTH	000 004	Read by others.
C_IWOTH	000 002	Write by others.
C_IXOTH	000 001	Execute by others.
C_ISUID	004 000	Set *uid*.
C_ISGID	002 000	Set *gid*.
C_ISVTX	001 000	Reserved.
File Type		
Name	**Value**	**Indicates**
C_ISDIR	040 000	Directory.
C_ISFIFO	010 000	FIFO.
C_ISREG	0100 000	Regular file.
C_ISBLK	060 000	Block special file.
C_ISCHR	020 000	Character special file.
C_ISCTG	0110 000	Reserved.
C_ISLNK	0120 000	Reserved.
C_ISSOCK	0140 000	Reserved.

Table 3-12 Values for *cpio* **c_mode** Field

Directories, FIFOs and regular files are supported on a system conforming to this document; additional values defined previously are reserved for compatibility with existing systems. Additional file types may be supported; however, such files should not be written to archives intended to be transported to other systems.

c_uid Contains the user ID of the owner.

c_gid Contains the group ID of the group.

c_nlink Contains the number of links referencing the file at the time the archive was created.

c_rdev Contains implementation-dependent information for character or block special files.

c_mtime
Contains the latest time of modification of the file at the time the archive was created.

c_namesize
Contains the length of the pathname, including the terminating NUL character.

c_filesize
Contains the length of the file in octets. This will be the length of the data section following the header structure.

The cpio Filename

The **c_name** field contains the pathname of the file. The length of this field in octets is the value of *c_namesize*. If a filename is found on the medium that would create an invalid pathname, it is implementation-dependent if the data from the file is stored on the file hierarchy and under what name it is stored.

All characters are represented in the ISO/IEC 646:1991 standard IRV. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users and groups, one or more implementation-dependent encodings of these characters will be provided for interchange purposes. However, the *pax* utility will never create filenames on the local system that cannot be accessed via the procedures described previously in this document. If a filename is found on the medium that would create an invalid filename, it is implementation-dependent if the data from the file is stored on the local file system and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being ignored.

The cpio File Data

Following *c_name*, there are **c_filesize** octets of data. Interpretation of such data occurs in a manner dependent on the file. If **c_filesize** is zero, no data will be contained in *c_filedata*.

When restoring from an archive:

- If the user does not have the appropriate privilege to create a file of the specified type, *pax* will ignore the entry and write an error message to standard error.
- Only regular files have data to be restored. Presuming a regular file meets any selection criteria that might be imposed on the format-reading utility by the user, such data will be restored.
- If a user does not have appropriate privilege to set a particular mode flag, the flag will be ignored. Some of the mode flags in the archive format are not mentioned elsewhere in this document. If the implementation does not support those flags, they may be ignored.

The cpio Special Entries

FIFO special files, directories and the trailer are recorded with **c_filesize** equal to zero. For other special files, **c_filesize** is unspecified by this document. The header for the next file entry in the archive will be written directly after the last octet of the file entry preceding it. A header denoting the filename TRAILER!!! indicates the end of the archive; the contents of octets in the last block of the archive following such a header are undefined.

Extended tar Format

An extended *tar* archive file contains a series of blocks. Each block will be a fixed-size block of 512 octets (see below). Each file archived is represented by a header block that describes the file, followed by zero or more blocks that give the contents of the file. At the end of the archive file there will be two blocks filled with binary zeros, interpreted as an end-of-archive indicator.

The blocks may be grouped for physical I/O operations. Each group of *n* blocks (where *n* is set by the *pax* –**b** option) may be written with a single write operation. On magnetic tape, the result of this write will be a single tape record. The last group of blocks always will be at the full size, so blocks after the two zero blocks may contain undefined data.

The header block will be structured as shown in the following table. All lengths and offsets are in decimal.

Field Name	Octet Offset	Length (in octets)
name	0	100
mode	100	8
uid	108	8
gid	116	8
size	124	12
mtime	136	12
chksum	148	8
typeflag	156	1
linkname	157	100
magic	257	6
version	263	2
uname	265	32
gname	297	32
devmajor	329	8
devminor	337	8
prefix	345	155

Table 3-13 Extended *tar* Header Block

All characters in the header block are represented in the coded character set of the ISO/IEC 646:1991 standard. For maximum portability between implementations, names should be selected from characters represented by the portable filename character set as octets with the most significant bit zero. If an implementation supports the use of characters outside the portable filename character set in names for files, users and groups, one or more implementation-dependent encodings of these characters will be provided for interchange purposes. However, the *pax* utility will never create filenames on the local system that cannot be accessed via the procedures described previously in this document. If a filename is found on the medium that would create an invalid filename, it is implementation-dependent if the data from the file is stored on the file hierarchy and under what name it is stored. The *pax* utility may choose to ignore these files as long as it produces an error indicating that the file is being

ignored.

Each field within the header block is contiguous; that is, there is no padding used. Each character on the archive medium is stored contiguously.

The fields **magic**, **uname** and **gname** are character strings each terminated by a NUL character. The fields **name**, **linkname** and **prefix** are NUL-terminated character strings except when all characters in the array contain non-NUL characters including the last character. The **version** field is two octets containing the characters **00** (zero-zero). The *typeflag* contains a single character. All other fields are leading zero-filled octal numbers using digits from the ISO/IEC 646:1991 standard IRV. Each numeric field is terminated by one or more space or NUL characters.

The **name** and the **prefix** fields produce the pathname of the file. The hierarchical relationship of the file can be retained by specifying the pathname as a path prefix, and a slash character and filename as the suffix. A new pathname is formed, if *prefix* is not an empty string (its first character is not NUL), by concatenating *prefix* (up to the first NUL character), a slash character and *name*; otherwise, *name* is used alone. In either case, *name* is terminated at the first NUL character. If *prefix* begins with a NUL character, it will be ignored. In this manner, pathnames of at most 256 characters can be supported. If a pathname does not fit in the space provided, *pax* will notify the user of the error, and will not store any part of the file header or data on the medium.

The **linkname** field, described below, does not use the *prefix* to produce a pathname. As such, a *linkname* is limited to 100 characters. If the name does not fit in the space provided, *pax* will notify the user of the error, and will not attempt to store the link on the medium.

The **mode** field provides 12 bits encoded in the ISO/IEC 646:1991 standard octal digit representation. The encoded bits represent the following values:

04 000 Set UID on execution.
02 000 Set GID on execution.
01 000 Reserved.
00 400 Read by owner.
00 200 Write by owner.
00 100 Execute/search by owner.
00 040 Read by group.
00 020 Write by group.
00 010 Execute/search by group.
00 004 Read by other.
00 002 Write by other.
00 001 Execute/search by other.

When appropriate privilege is required to set one of these mode bits, and the user restoring the files from the archive does not have the appropriate privilege, the mode bits for which the user does not have appropriate privilege will be ignored. Some of the mode bits in the archive format are not mentioned elsewhere in this document. If the implementation does not support those bits, they may be ignored.

The **uid** and **gid** fields are the user and group ID of the owner and group of the file, respectively.

The **size** field is the size of the file in octets. If the **typeflag** field is set to specify a file to be of type 1 (a link) or 2 (reserved for symbolic links), the **size** field will be specified as zero. If the **typeflag** field is set to specify a file of type 5 (directory), the **size** field will be interpreted as described under the definition of that record type. No data blocks will be stored for types 1, 2 or 5. If the **typeflag** field is set to 3 (character special file), 4 (block special file), or 6 (FIFO), the meaning of the **size** field is unspecified by this document, and no data blocks will be stored on

the medium. Additionally, for 6, the **size** field is ignored when reading. If the **typeflag** field is set to any other value, the number of blocks written following the header will be (**size**+511)/512, ignoring any fraction in the result of the division.

The **mtime** field is the modification time of the file at the time it was archived. It is the ISO/IEC 646:1991 standard representation of the octal value of the modification time obtained from the **XSH** specification *stat*( ) function.

The **chksum** field is the ISO/IEC 646:1991 standard IRV representation of the octal value of the simple sum of all octets in the header block. Each octet in the header is treated as an unsigned value. These values will be added to an unsigned integer, initialised to zero, the precision of which will be not less than 17 bits. When calculating the checksum, the **chksum** field is treated as if it were all spaces.

The **typeflag** field specifies the type of file archived. If a particular implementation does not recognise the type, or the user does not have appropriate privilege to create that type, the file will be extracted as if it were a regular file if the file type is defined to have a meaning for the **size** field that could cause data blocks to be written on the medium (see the previous description for *size*). If conversion to a regular file occurs, the *pax* utility will produce an error indicating that the conversion took place. All of the **typeflag** fields will be coded in the ISO/IEC 646:1991 standard IRV:

'0' Represents a regular file. For backward compatibility, a **typeflag** value of binary zero ('\0') should be recognised as meaning a regular file when extracting files from the archive. Archives written with this version of the archive file format create regular files with a **typeflag** value of the ISO/IEC 646:1991 standard IRV 0.

'1' Represents a file linked to another file, of any type, previously archived. Such files are identified by each file having the same device and file serial number. The linked-to name is specified in the **linkname** field with a NUL-character terminator if it is less than 100 octets in length.

'2' Reserved to represent a link to another file, of any type, whose device or file serial number differs. This is provided for systems that support linked files whose device or file serial numbers differ, and should be treated as a type 1 file if this extension does not exist.

'3','4' Represent character special files and block special files respectively. In this case the **devmajor** and **devminor** fields contain information defining the device, the format of which is unspecified by this document. Implementations may map the device specifications to their own local specification or may ignore the entry.

'5' Specifies a directory or subdirectory. On systems where disk allocation is performed on a directory basis, the **size** field will contain the maximum number of octets (which may be rounded to the nearest disk block allocation unit) that the directory may hold. A **size** field of zero indicates no such limiting. Systems that do not support limiting in this manner should ignore the **size** field.

'6' Specifies a FIFO special file. Note that the archiving of a FIFO file archives the existence of this file and not its contents.

'7' Reserved to represent a file to which an implementation has associated some high-performance attribute. Implementations without such extensions should treat this file as a regular file (type 0).

'A'–'Z' The letters A to Z, inclusive, are reserved for custom implementations. All other values are reserved for specification in future issues.

The **magic** field is the specification that this archive was output in this archive format. If this field contains **ustar** (the five characters from the ISO/IEC 646:1991 standard IRV shown followed by NUL), the **uname** and **gname** fields will contain the ISO/IEC 646:1991 standard IRV representation of the owner and group of the file respectively (truncated to fit, if necessary). When the file is restored by a privileged, protection-preserving version of the utility, the password and group files will be scanned for these names. If found, the user and group IDs contained within these files will be used rather than the values contained within the **uid** and **gid** fields.

EXIT STATUS

The following exit values are returned:

0 All files were processed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If *pax* cannot create a file or a link when reading an archive or cannot find a file when writing an archive, or cannot preserve the user ID, group ID or file mode when the **–p** option is specified, a diagnostic message will be written to standard error and a non-zero exit status will be returned, but processing will continue. In the case where *pax* cannot create a link to a file, *pax* will not, by default, create a second copy of the file.

If the extraction of a file from an archive is prematurely terminated by a signal or error, *pax* may have only partially extracted the file or (if the **–n** option was not specified) may have extracted a file of the same name as that specified by the user, but which is not the file the user wanted. Additionally, the file modes of extracted directories may have additional bits from the S_IRWXU mask set as well as incorrect modification and access times.

APPLICATION USAGE

The **–p** (privileges) option was invented to reconcile differences between historical *tar* and *cpio* implementations. In particular, the two utilities use **–m** in diametrically opposed ways. The **–p** option also provides a consistent means of extending the ways in which future file attributes can be addressed, such as for enhanced security systems or high-performance files. Although it may seem complex, there are really two modes that will be most commonly used:

–p e ‘‘Preserve everything’’. This would be used by the historical superuser, someone with all the appropriate privileges, to preserve all aspects of the files as they are recorded in the archive. The **e** flag is the sum of **o** and **p**, and other implementation-dependent attributes.

–p p ‘‘Preserve’’ the file mode bits. This would be used by the user with regular privileges who wished to preserve aspects of the file other than the ownership. The file times are preserved by default, but two other flags are offered to disable these and use the time of extraction.

The one pathname per line format of standard input precludes pathnames containing newline characters. Although such pathnames violate the portable filename guidelines, they may exist and their presence may inhibit usage of *pax* within shell scripts. This problem is inherited from historical archive programs. The problem can be avoided by listing filename arguments on the command line instead of on standard input.

It is almost certain that appropriate privileges will be required for *pax* to accomplish parts of this specification. Specifically, creating files of type block special or character special, restoring file access times unless the files are owned by the user (the **–t** option) or preserving file owner, group, and mode (the **–p** option) will all probably require appropriate privileges.

In read mode, implementations are permitted to overwrite files when the archive has multiple members with the same name. This may fail if permissions on the first version of the file do not permit it to be overwritten.

EXAMPLES

The following command:

```
pax -w -f /dev/rmt/1m .
```

copies the contents of the current directory to tape drive 1, medium density (assuming historical System V device naming procedures. The historical BSD device name would be /dev/rmt9).

The following commands:

```
mkdir newdir
pax -rw olddir newdir
```

copy the *olddir* directory hierarchy to *newdir*.

```
pax -r -s ',^//*usr//*,,' -f a.pax
```

reads the archive **a.pax**, with all files rooted in **/usr** in the archive extracted relative to the current directory.

FUTURE DIRECTIONS

It is expected that future versions of the ISO/IEC 9945-2:1993 standard will offer additional file formats and the **–o** option will be used by future issues to specify such features as international filename and file codeset translations, security, accounting, and so forth, related to each additional format.

SEE ALSO

cpio, tar.

CHANGE HISTORY

First released in Issue 4.

NAME

pcat – expand and concatenate files (**TO BE WITHDRAWN**)

SYNOPSIS

EX `pcat` *`file...`*

DESCRIPTION

The *pcat* utility unpacks files in the format used by *pack* and writes the unpacked form to standard output. For each *file* operand, a file named *file***.z** (or just *file*, if *file* ends in **.z**) is unpacked.

A file is not written in its unpacked form if:

- The file cannot be opened.
- The file does not appear to be the output of *pack*.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of a file to be *pcat*ed; *file* can include or omit the **.z** suffix.

STDIN

Not used.

INPUT FILES

The input files are regular files in the format produced by the *pack* utility.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *pack*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is the concatenation of the unpacked files identified by the *file* operands.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *pcat* utility does for packed files what *cat* does for ordinary files, except that *pcat* cannot be used as a filter.

EXAMPLES

To view a packed file named *file***.z** use:

```
pcat file.z
```

or:

```
pcat file
```

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

```
pcat file >abc
```

FUTURE DIRECTIONS

This utility is being withdrawn from a future issue. The *zcat* utility should be used instead.

SEE ALSO

pack, unpack, zcat.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Split into a separate description.

Marked **TO BE WITHDRAWN**.

Internationalised environment variable support made optional.

Issue 4, Version 2

An assertion formerly in the **DESCRIPTION**, that a file is not written if the filename has more than {NAME_MAX}–2 bytes, is deleted.

NAME

pg – file perusal filter for soft-copy terminals (**TO BE WITHDRAWN**)

SYNOPSIS

EX `pg[-number][-cefns][-p string][+linenumber][+/re/][file...]`

DESCRIPTION

The *pg* utility is a filter that allows the examination of the named *file* or files one screenful at a time on a soft-copy terminal. Each screenful is followed by a prompt. If the user types a carriage-return character, another page is displayed; other possibilities are enumerated in **EXTENDED DESCRIPTION**.

If the standard output is not a terminal, *pg* ignores all options and writes the input files to standard output, like the *cat* utility, except that a header is written before each file (if there is more than one).

OPTIONS

The *pg* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the –*number* option takes a multi-digit value and the +*linenumber* and +/*re*/ options take a leading plus sign instead of minus. The following options are supported when standard input is a terminal:

–*number*
: Change window size. The *number* argument is a positive integer specifying the size (in lines) of the window that *pg* is to use instead of the default. (The default size is determined by the *LINES* environment variable. If *LINES* is unset or null, a terminal-specific default size is used; the *TERM* environment variable may affect the determination of the terminal-specific default size. The default is typically one less than the number of lines in the terminal window.)

–**p** *string*
: 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 the terminal does not support this.

–**e** Do not pause at the end of each file; see the **EXTENDED DESCRIPTION**.

–**f** Inhibit line splitting. Normally, *pg* splits lines longer than the screen width, but some sequences of characters in the text being displayed (for example, escape sequences for underlining) generate undesirable results.

UN –**n** Cause an automatic end-of-command as soon as a command letter is entered. Normally, commands must be terminated by a newline character. The –**n** option is not supported on certain block-mode terminals. If *pg* is being used on such terminal, the –**n** option will be silently ignored and the terminating newline character will be required to complete a command.

–**s** Write all messages and prompts in standout mode (usually inverse video) if the terminal supports this.

+*linenumber*
: Start up at *linenumber*.

+/*re*/ Start up at the first line matching the basic regular expression *re*.

OPERANDS

The following operands are supported:

file A pathname of a text file to be displayed. If no *file* is given, or if it is –, the standard input is read.

STDIN

The standard input is a text file that is used if no *file* operand is given, or if it is –.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *pg*:

COLUMNS
Determine the horizontal screen size. If unset or null, use the value of *TERM*, the window size, baud rate, or some combination of these, to indicate the terminal type for the screen size calculation.

LINES Determine the number of lines to be displayed on the screen. If unset or null, use the value of *TERM*, the window size, baud rate, or some combination of these, to indicate the terminal type for the screen size calculation.

SHELL Determine the name of the command interpreter executed for a !command.

TERM Determine terminal attributes. Optionally attempt to search a system-dependent database, keyed on the value of the *TERM* environment variable. If no information is available, a terminal incapable of cursor-addressable movement is assumed.

The following environment variables may affect the execution of *pg*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within regular expressions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

During the writing of characters to the standard output, *pg* catches SIGINT and SIGQUIT,

returning the user to the prompt. When waiting at the prompt, *pg* takes the default action on SIGINT and SIGQUIT. The default actions are taken for all other signals.

STDOUT

The standard output is a display of the input files, with prompts interspersed. When standard output is not associated with a terminal, it consists of the contents of the input files, separated by the following lines, if there is more than one input file:

```
"::::::::::::::\n%s\n::::::::::::::\n", <pathname>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The responses that can 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 that 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 <space>
: Display one page. The address is specified in number of pages.

(+1)**l**
: With a relative address, simulate scrolling the screen, forward or backward, by the number of lines specified. With an absolute address, print a screenful beginning at the specified line.

(+1)**d** or <control>-D
: Simulate scrolling half a screen forward or backward.

The following perusal commands take no address:

. or <control>-L
: Redisplay the current page of text.

$
: Display the last windowful in the file.

The following commands are available for searching for text patterns in the text. Basic regular expression syntax is used in patterns. The *re*s must always be terminated by a newline character, even if the **–n** option is specified.

i/*re*/
: Search forwards for the *i*th occurrence of *re* (default *i*=1). Searching begins immediately after the current page and continues to the end of the current file, without wrap-around.

i^*re*^
i?*re*?
: Search backwards for the *i*th occurrence of *re* (default *i*=1). Searching begins immediately before the current page and continues to the beginning of the current file, without wrap-around.

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.

If the standard output is not a terminal device, *pg* always exits when it reaches end-of-file on the last file in its argument list. Otherwise, for all files but the last, *pg* prompts with an indication that it has reached the end of file, along with the name of the next file; when **–e** is used, the end-of-file prompt is bypassed and the prompt indicates the name of the next file. For the last file specified, or for the standard input if no file is specified, *pg* prompts indicating end-of-file (if **–e** is not used), and accepts additional commands. If the next command specifies forward scrolling, *pg* exits. If the **–e** option is specified, *pg* exits immediately after writing the last line of the last file.

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

i **n** Begin perusing the *i*th next file in the command line. The *i* is an unsigned number; default value is 1.

i **p** Begin perusing the *i*th previous file in the command line. The *i* is an unsigned number; default is 1.

i **w** Display another window of text. If *i* is present, set the window size to *i*.

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

h Help by displaying an abbreviated summary of available commands.

q or **Q** Quit *pg*.

!*command*
Pass the argument *command* to the command interpreter, whose name is taken from the *SHELL* environment variable. If *SHELL* is unset or null, *sh* is used as the default command interpreter. This command must always be terminated by a newline character, even if the **–n** option is specified.

At any time when output is being sent to the terminal, receipt of a quit or interrupt signal 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 signal occurs.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility 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 in **EXTENDED DESCRIPTION**.

While waiting for terminal input, *pg* responds to SIGINT and SIGQUIT signals 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.

Use the **$** command with caution when the input is a pipe.

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

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

EXAMPLES

None.

FUTURE DIRECTIONS

This utility will be withdrawn from a future issue. The *more* utility should be used instead. Modifications to achieve full conformance to the Utility Syntax Guidelines were not made.

SEE ALSO

more, the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**, the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable and regular expression support made optional.

Marked **TO BE WITHDRAWN**.

NAME

pr – print files

SYNOPSIS

EX

```
pr [+page][-c column][-adFmrt][-e[char][gap]][-h header][-i[char][gap]]
[-l lines][-n[char][width]][-o offset][-s[char]][-w width][-fp]
[file...]
```

DESCRIPTION

The *pr* utility is a printing and pagination filter. If multiple input files are specified, each is read, formatted, and written to standard output. By default, the input is separated into 66-line pages, each with:

- a 5-line header that includes the page number, date, time and the pathname of the file
- a 5-line trailer consisting of blank lines.

If standard output is associated with a terminal, diagnostic messages will be deferred until the *pr* utility has completed processing.

When options specifying multi-column output are specified, output text columns will be of equal width; input lines that do not fit into a text column will be truncated. By default, text columns are separated with at least one blank character.

OPTIONS

The *pr* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that: the *page* option has a + delimiter; *page* and *column* can be multi-digit numbers; some of the option-arguments are optional; and some of the option-arguments cannot be specified as separate arguments from the preceding option letter. In particular, the **–s** option does not allow the option letter to be separated from its argument, and the options **–e**, **–i** and **–n** require that both arguments, if present, not be separated from the option letter.

The following options are supported. In the following option descriptions, *column*, *lines*, *offset*, *page* and *width* are positive decimal integers; *gap* is a non-negative decimal integer.

+page Begin output at page number *page* of the formatted input.

–column Produce multi-column output that is arranged in *column* columns (default is 1) and is written down each column in the order in which the text is received from the input file. This option should not be used with **–m**. The options **–e** and **–i** will be assumed for multiple text-column output. Whether or not text columns are produced with identical vertical lengths is unspecified, but a text column will never exceed the length of the page (see the **–l** option). When used with **–t**, use the minimum number of lines to write the output.

–a Modify the effect of the *–column* option so that the columns are filled across the page in a round-robin order (for example, when *column* is 2, the first input line heads column 1, the second heads column 2, the third is the second line in column 1, and so forth).

–d Produce output that is double-spaced; append an extra newline character following every newline character found in the input.

–e[*char***][***gap***]**
Expand each input tab character to the next greater column position specified by the formula *n***gap*+1, where *n* is an integer > 0. If *gap* is zero or is omitted, it defaults to 8. All tab characters in the input will be expanded into the appropriate number of space characters. If any non-digit character, *char*, is specified, it will be used as the input tab character.

EX **–f** Use a form-feed character for new pages, instead of the default behaviour that uses a sequence of newline characters. Pause before beginning the first page if the standard output is associated with a terminal.

–F Use a form-feed character for new pages, instead of the default behaviour that uses a sequence of newline characters.

–h *header*
Use the string *header* to replace the contents of the *file* operand in the page header.

–i[*char***][***gap***]**
In output, replace multiple space characters with tab characters wherever two or more adjacent space characters reach column positions *gap*+1, 2**gap*+1, 3**gap*+1, and so forth. If *gap* is zero or is omitted, default tab settings at every eighth column position are assumed. If any non-digit character, *char*, is specified, it will be used as the output tab character.

–l *lines* Override the 66-line default and reset the page length to *lines*. If *lines* is not greater than the sum of both the header and trailer depths (in lines), the *pr* utility will suppress both the header and trailer, as if the **–t** option were in effect.

–m Merge files. Standard output will be formatted so the *pr* utility writes one line from each file specified by a *file* operand, side by side into text columns of equal fixed widths, in terms of the number of column positions. Implementations support merging of at least nine *file* operands.

–n[*char***][***width***]**
Provide *width*-digit line numbering (default for *width* is 5). The number will occupy the first *width* column positions of each text column of default output or each line of **–m** output. If *char* (any non-digit character) is given, it will be appended to the line number to separate it from whatever follows (default for *char* is a tab character).

–o *offset*
Each line of output will be preceded by offset <space>s. If the **–o** option is not specified, the default offset is zero. The space taken will be in addition to the output line width (see **–w** option below).

EX **–p** Pause before beginning each page if the standard output is directed to a terminal (*pr* will write an alert character to standard error and wait for a carriage-return character to be read on **/dev/tty**).

–r Write no diagnostic reports on failure to open files.

–s[*char***]**
Separate text columns by the single character *char* instead of by the appropriate number of space characters (default for *char* is the tab character).

–t Write neither the five-line identifying header nor the five-line trailer usually supplied for each page. Quit writing after the last line of each file without spacing to the end of the page.

–w *width*
Set the width of the line to *width* column positions for multiple text-column output only. If the **–w** option is not specified and the **–s** option is not specified, the default width is 72. If the **–w** option is not specified and the **–s** option is specified, the default width is 512.

For single column output, input lines will not be truncated.

OPERANDS

The following operand is supported:

file A pathname of a file to be written. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files.

EX The file **/dev/tty** is used to read responses required by the –**p** option.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *pr*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments and input files) and which characters are defined as printable (character class **print**). Non-printable characters still will be written to standard output, but are not counted for the purpose for column-width and line-length calculations.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME

Determine the format of the date and time for use in writing header lines.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone for use in writing header lines.

ASYNCHRONOUS EVENTS

If *pr* receives an interrupt while writing to a terminal, it will flush all accumulated error messages to the screen before terminating.

STDOUT

The *pr* utility output will be a paginated version of the original file (or files). This pagination will be accomplished using either form-feed characters or a sequence of newline characters, as EX controlled by the –**F** or –**f** option. Page headers will be generated unless the –**t** option is specified. The page headers will be of the form:

```
"\n\n%s %s Page %d\n\n\n", <output of date>, <file>, <page number>
```

In the POSIX locale, the <*output of date*> field, representing the date and time of last modification of the input file (or the current date and time if the input file is standard input), is equivalent to

the output of the following command as it would appear if executed at the given time:

```
date "+%b %e %H:%M %Y"
```

without the trailing newline character, if the page being written is from standard input. If the page being written is not from standard input, in the POSIX locale, the same format will be used, but the time used will be the modification time of the file corresponding to *file* instead of the current time. When the LC_TIME locale category is not set to the POSIX locale, a different format and order of presentation of this field may be used.

If the standard input is used instead of a *file* operand, the <*file*> field will be replaced by a null string.

If the **–h** option is specified, the <*file*> field will be replaced by the *header* argument.

STDERR

EX Used for diagnostic messages and for alerting the terminal when **–p** is specified.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All files were written successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. Print a numbered list of all files in the current directory:

   ```
   ls -a | pr -n -h "Files in $(pwd)."
   ```

2. Print **file1** and **file2** as a double-spaced, three-column listing headed by ‘‘file list’’:

   ```
   pr -3d -h "file list" file1 file2
   ```

3. Write **file1** on **file2**, expanding tabs to columns 10, 19, 28, . . . :

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

FUTURE DIRECTIONS

It is possible that a new interface that conforms to the Utility Syntax Guidelines will be introduced.

SEE ALSO

expand, *lp*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

printf – write formatted output

SYNOPSIS

```
printf format[argument...]
```

DESCRIPTION

The *printf* utility will write formatted operands to the standard output. The *argument* operands will be formatted under control of the *format* operand.

OPTIONS

None.

OPERANDS

The following operands are supported:

format A string describing the format to use to write the remaining operands; see **EXTENDED DESCRIPTION**.

argument
The strings to be written to standard output, under the control of *format*; see **EXTENDED DESCRIPTION**.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *printf*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_NUMERIC
Determine the locale for numeric formatting. It will affect the format of numbers written using the e, E, f, g and G conversion characters (if supported).

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See **EXTENDED DESCRIPTION**.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

The *format* operand will be used as the *format* string described in the **XBD** specification, **Chapter 3**, **File Format Notation** with the following exceptions:

- A space character in the format string, in any context other than a flag of a conversion specification, will be treated as an ordinary character that is copied to the output.
- A Δ character in the format string will be treated as a Δ character, not as a space character.
- In addition to the escape sequences shown in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v), *****ddd*, where *ddd* is a one-, two- or three-digit octal number, will be written as a byte with the numeric value specified by the octal number.
- The implementation will not precede or follow output from the d or u conversion specifications with blank characters not specified by the *format* operand.
- The implementation will not precede output from the o conversion specification with zeros not specified by the *format* operand.
- The e, E, f, g and G conversion specifications need not be supported.
- An additional conversion character, b, will be supported as follows. The argument will be taken to be a string that may contain backslash-escape sequences. The following backslash-escape sequences will be supported:

  — the escape sequences listed in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v), which will be converted to the characters they represent

  — **\0***ddd*, where *ddd* is a zero-, one-, two- or three-digit octal number that will be converted to a byte with the numeric value specified by the octal number

  — \c, which will not be written and will cause *printf* to ignore any remaining characters in the string operand containing it, any remaining string operands and any additional characters in the *format* operand.

  The interpretation of a backslash followed by any other sequence of characters is unspecified.

  Bytes from the converted string will be written until the end of the string or the number of bytes indicated by the precision specification is reached. If the precision is omitted, it will be taken to be infinite, so all bytes up to the end of the converted string will be written.
- For each specification that consumes an argument, the next argument operand will be evaluated and converted to the appropriate type for the conversion as specified below.
- The *format* operand will be reused as often as necessary to satisfy the argument operands. Any extra c or s conversion specifications will be evaluated as if a null string argument were supplied; other extra conversion specifications will be evaluated as if a zero argument were supplied. If the *format* operand contains no conversion specifications and *argument* operands are present, the results are unspecified.
- If a character sequence in the *format* operand begins with a % character, but does not form a valid conversion specification, the behaviour is unspecified.

The *argument* operands will be treated as strings if the corresponding conversion character is b, c or s; otherwise, it will be evaluated as a C constant, as described by the ISO C standard, with the following extensions:

- A leading plus or minus sign will be allowed.
- If the leading character is a single- or double-quote, the value will be the numeric value in the underlying codeset of the character following the single- or double-quote.

If an argument operand cannot be completely converted into an internal value appropriate to the corresponding conversion specification, a diagnostic message will be written to standard error and the utility will not exit with a zero exit status, but will continue processing any remaining operands and will write the value accumulated at the time the error was detected to standard output.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The floating-point formatting conversion specifications of *printf*( ) are not required because all arithmetic in the shell is integer arithmetic. The *awk* utility performs floating-point calculations and provides its own **printf** function. The *bc* utility can perform arbitrary-precision floating-point arithmetic, but does not provide extensive formatting capabilities. (This *printf* utility cannot really be used to format *bc* output; it does not support arbitrary precision.) Implementations are encouraged to support the floating-point conversions as an extension.

Note that this *printf* utility, like the **XSH** specification *printf*( ) function on which it is based, makes no special provision for dealing with multi-byte characters when using the **%c** conversion specification or when a precision is specified in a **%b** or **%s** conversion specification. Applications should be extremely cautious using either of these features when there are multi-byte characters in the character set.

Field widths and precisions cannot be specified as * since the * can be replaced directly in the *format* operand using shell variable substitution. Implementations can also provide this feature as an extension if they so choose.

Hexadecimal character constants as defined in the ISO C standard are not recognised in the *format* operand because there is no consistent way to detect the end of the constant. Octal character constants are limited to, at most, three octal digits, but hexadecimal character constants are only terminated by a non-hex-digit character. In the ISO C standard, the ## concatenation operator can be used to terminate a constant and follow it with a hexadecimal character to be written. In the shell, concatenation occurs before the *printf* utility has a chance to parse the end of the hexadecimal constant.

The **%b** conversion specification is not part of the ISO C standard; it has been added here as a portable way to process backslash escapes expanded in string operands as provided by the *echo* utility. See also the **APPLICATION USAGE** section of *echo* for ways to use *printf* as a replacement for all of the traditional versions of the *echo* utility.

If an argument cannot be parsed correctly for the corresponding conversion specification, the *printf* utility is required to report an error. Thus, overflow and extraneous characters at the end of an argument being used for a numeric conversion are to be reported as errors.

It is not considered an error if an argument operand is not completely used for a c or s conversion or if a string operand's first or second character is used to get the numeric value of a character.

EXAMPLES

To alert the user and then print and read a series of prompts:

```
printf "\aPlease fill in the following: \nName: "
read name
printf "Phone number: "
read phone
```

To read out a list of right and wrong answers from a file, calculate the percentage correctly, and print them out. The numbers are right-justified and separated by a single tab character. The percentage is written to one decimal place of accuracy:

```
while read right wrong ; do
   percent=$(echo "scale=1;($right*100)/($right+$wrong)" | bc)
   printf "%2d right\t%2d wrong\t(%s%%)\n" \
          $right $wrong $percent
done < database_file
```

The command:

```
printf "%5d%4d\n" 1 21 321 4321 54321
```

produces:

```
    1  21
  3214321
54321   0
```

Note that the *format* operand is used three times to print all of the given strings and that a 0 was supplied by *printf* to satisfy the last **%4d** conversion specification.

The *printf* utility is required to notify the user when conversion errors are detected while producing numeric output; thus, the following results would be expected on an implementation with 32-bit twos-complement integers when **%d** is specified as the *format* operand:

Argument	Standard Output	Diagnostic Output
5a	5	**printf: "5a" not completely converted**
9999999999	2147483647	**printf: "9999999999" arithmetic overflow**
-9999999999	-2147483648	**printf: "-9999999999" arithmetic overflow**
ABC	0	**printf: "ABC" expected numeric value**

The diagnostic message format is not specified, but these examples convey the type of information that should be reported. Note that the value shown on standard output is what would be expected as the return value from the **XSH** specification function *strtol*( ). A similar correspondence exists between **%u** and *strtoul*( ) and **%e**, **%f** and **%g** (if the implementation supports floating-point conversions) and *strtod*( ).

In a locale using the ISO/IEC 646:1991 standard as the underlying codeset, the command:

```
printf "%d\n" 3 +3 -3 \'3 \"+3 "'-3"
```

produces:

3	Numeric value of constant 3
3	Numeric value of constant 3
–3	Numeric value of constant –3
51	Numeric value of the character '3' in the ISO/IEC 646:1991 standard codeset
43	Numeric value of the character '+' in the ISO/IEC 646:1991 standard codeset
45	Numeric value of the character '–' in the ISO/IEC 646:1991 standard codeset

Note that in a locale with multi-byte characters, the value of a character is intended to be the value of the equivalent of the **wchar_t** representation of the character as described in the **XSH** specification.

FUTURE DIRECTIONS

None.

SEE ALSO

awk, *bc*, *echo*, the **XSH** specification description of *printf*( ).

CHANGE HISTORY

First released in Issue 4.

NAME

prs – print an SCCS file (**DEVELOPMENT**)

SYNOPSIS

EX `prs [-a][-d dataspec][-r[SID]] file...`

EX `prs [ -e| -l] -c cutoff [-d dataspec] file...`

EX `prs [ -e| -l] -r[SID][-d dataspec]file...`

DESCRIPTION

The *prs* utility writes to standard output parts or all of an SCCS file in a user-supplied format.

OPTIONS

The *prs* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the **–r** option has an optional option-argument. This optional option-argument cannot be presented as a separate argument. The following options are supported:

–d *dataspec*
Specify the output data specification. The *dataspec* is a string consisting of SCCS file *data keywords* (see **Data Keywords** on page 589) interspersed with optional user-supplied text.

–r[*SID***]**
Specify the SCCS identification string (SID) of a delta for which information is desired. If no *SID* option-argument is specified, the SID of the most recently created delta is assumed.

–e Request information for all deltas created earlier than and including the delta designated via the **–r** option or the date-time given by the **–c** option.

–l Request information for all deltas created later than and including the delta designated via the **–r** option or the date-time given by the **–c** option.

–c *cutoff*
Indicate the *cutoff* date-time, in the form:

YY[*MM*[*DD*[*HH*[*MM*[*SS*]]]]]

No changes (deltas) to the SCCS file that were created after the specified *cutoff* date-time are included in the output. Units omitted from the date-time default to their maximum possible values; for example, **–c 7502** is equivalent to **–c 750228235959**.

–a Request writing of information for both removed, that is, delta type = *R* (see *rmdel*) and existing, that is, delta type = *D*, deltas. If the **–a** option is not specified, information for existing deltas only is provided.

OPERANDS

The following operand is supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *prs* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only when the *file* operand is specified as –. Each line of the text file is interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files displayed are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *prs*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file whose format is dependent on the data keywords specified with the **–d** option.

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. A data keyword may appear in a *dataspec* multiple times.

The information written by *prs* consists of: (1) the user-supplied text; and (2) appropriate values (extracted from the SCCS file) substituted for the recognised 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 newline.

User-supplied text is any text other than recognised data keywords. A tab character is specified by `\t` and newline by `\n`. When the **–r** option is not specified, the default *dataspec* is:

```
:PN::\n\n
```

and the following *dataspec* is used for each selected delta:

```
:Dt:\t:DL:\nMRs:\n:MR:COMMENTS:\n:C:\n
```

SCCS File 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)	"	See below**	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 included, excluded or 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

SCCS File Data Keywords

Keyword	Data Item	File Section	Value	Format
: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` string	N/A	**:Z::M:\t:I:**	S
:A:	A form of `what` string	N/A	**:Z::Y: :M: :I::Z:**	S
:Z:	`what` string delimiter	N/A	**@(#)**	S
:F:	SCCS filename	N/A	*text*	S
:PN:	SCCS file pathname	N/A	*text*	S

* **:Dt: = :DT: :I: :D: :T: :P: :DS: :DP:**

** **:R:.:L:.:B:.:S:** if the delta is a branch delta (`:BF:` == yes)
:R:.:L: if the delta is not a branch delta (`:BF:` == no)

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. The following example:

   ```
   prs –d "User Names for :F: are:\n:UN:" s.file
   ```

   may write to standard output:

   ```
   User Names for s.file are:
   xyz
   131
   abc
   ```

2. The following example:

   ```
   prs –d "Delta for pgm :M:: :I: - :D: By :P:" –r s.file
   ```

   may write to standard output:

   ```
   Delta for pgm main.c: 3.7 - 77/12/01 By cas
   ```

3. As a special case:

   ```
   prs s.file
   ```

   may write to standard output:

   ```
   s.file:
   <blank line>
   D 1.1 77/12/01 00:00:00 cas 1 000000/00000/00000
   MRs:
   bl78-12345
   bl79-54321
   COMMENTS:
   this is the comment line for s.file initial delta
   <blank line>
   ```

   for each delta table entry of the *D* type. The only option allowed to be used with this special case is the **–a** option.

FUTURE DIRECTIONS

A version of *prs* that fully supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** may be introduced in a future issue.

SEE ALSO

admin, *delta*, *get*, *what*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

NAME

ps – report process status

SYNOPSIS

EX `ps [-aA][-defl][-G grouplit][-o format]...[-p proclist][-t termlist]`
EX `[-U userlist][-g grouplist][-nnamlist][-u userlist]`

DESCRIPTION

The *ps* utility writes information about processes, subject to having the appropriate privileges to obtain information about those processes.

By default, *ps* selects all processes with the same effective user ID as the current user and the same controlling terminal as the invoker.

OPTIONS

The *ps* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Write information for all processes associated with terminals. Implementations may omit session leaders from this list.

–A Write information for all processes.

EX **–d** Write information for all processes, except session leaders.

EX **–e** Write information for all processes (equivalent to **–A**).

EX **–f** Generate a *full* listing. (See **STDOUT** for the contents of a full listing.)

EX **–g** *grouplist*
Write information for processes whose session leaders are given in *grouplist*. The *grouplist* must be a single argument in the form of a blank- or comma-separated list.

–G *grouplist*
Write information for processes whose real group ID numbers are given in *grouplist*. The *grouplist* must be a single argument in the form of a blank- or comma-separated list.

EX **–l** Generate a *long* listing. (See **STDOUT** for the contents of a long listing.)

EX **–n** *namelist*
Specify the name of an alternative system *namelist* file in place of the default. The name of the default file and the format of a *namelist* file are unspecified.

–o *format*
Write information according to the format specification given in *format*. This is fully described in **STDOUT**. Multiple **–o** options can be specified; the format specification will be interpreted as the space-character-separated concatenation of all the *format* option-arguments.

–p *proclist*
Write information for processes whose process ID numbers are given in *proclist*. The *proclist* must be a single argument in the form of a blank- or comma-separated list.

–t *termlist*
Write information for processes associated with terminals given in *termlist*. The *termlist* must be a single argument in the form of a blank- or comma-separated list. Terminal
EX identifiers must be given in one of two forms: the device's filename (for example, **tty04**) or, if the device's filename starts with **tty**, just the identifier following the characters **tty** (for example, **04**).

EX **–u** *userlist*
Write information for processes whose user ID numbers or login names are given in *userlist*. The *userlist* must be a single argument in the form of a blank- or comma-separated list. In the listing, the numerical user ID will be written unless the **–f** option is used, in which case the login name will be written.

–U *userlist*
Write information for processes whose real user ID numbers or login names are given in *userlist*. The *userlist* must be a single argument in the form of a blank- or comma-separated list.

With the exception of **–o** *format*, all of the options shown are used to select processes. If any are specified, the default list will be ignored and *ps* will select the processes represented by the inclusive OR of all the selection-criteria options.

OPERANDS
None.

STDIN
Not used.

INPUT FILES
None.

ENVIRONMENT VARIABLES
The following environment variables affect the execution of *ps*:

COLUMNS
Override the system-selected horizontal screen size, used to determine the number of text columns to display. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

LC_TIME
Determine the format and contents of the date and time strings displayed.

ASYNCHRONOUS EVENTS
Default.

STDOUT

EX When the –**o** option is not specified, the standard output format is as follows. The column headings and descriptions of the columns in a *ps* listing are given below. The precise meanings of these fields are implementation-dependent. The letters **f** and **l** (below) 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 this datum is known.
PPID	(f,l)	The process ID of the parent process.
C	(f,l)	Processor utilisation 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 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 written 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 written in square brackets.

The –**o** option allows the output format to be specified under user control.

The format specification must be a list of names presented as a single argument, blank- or comma-separated. Each variable has a default header. The default header can be overridden by appending an equals sign and the new text of the header. The rest of the characters in the argument will be used as the header text. The fields specified will be written in the order specified on the command line, and should be arranged in columns in the output. The field widths will be selected by the system to be at least as wide as the header text (default or overridden value). If the header text is null, such as –**o** user=, the field width will be at least as wide as the default header text. If all header text fields are null, no header line will be written.

The following names are recognised in the POSIX locale:

ruser The real user ID of the process. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise.

user The effective user ID of the process. This will be the textual user ID, if it can be obtained and the field width permits, or a decimal representation otherwise.

rgroup The real group ID of the process. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise.

group The effective group ID of the process. This will be the textual group ID, if it can be obtained and the field width permits, or a decimal representation otherwise.

pid The decimal value of the process ID.

ppid The decimal value of the parent process ID.

pgid The decimal value of the process group ID.

pcpu The ratio of CPU time used recently to CPU time available in the same period, expressed as a percentage. The meaning of ''recently'' in this context is unspecified. The CPU time available is determined in an unspecified manner.

vsz The size of the process in (virtual) memory in kilobytes as a decimal integer.

nice The decimal value of the system scheduling priority of the process. See *nice*.

etime In the POSIX locale, the elapsed time since the process was started, in the form:

[[*dd***-]***hh***:]***mm***:***ss*

where *dd* will represent the number of days, *hh* the number of hours, *mm* the number of minutes, and *ss* the number of seconds. The *dd* field will be a decimal integer. The *hh*, *mm* and *ss* fields will be two-digit decimal integers padded on the left with zeros.

time In the POSIX locale, the cumulative CPU time of the process in the form:

[*dd***-]***hh***:***mm***:***ss*

The *dd*, *hh*, *mm* and *ss* fields will be as described in the **etime** specifier.

tty The name of the controlling terminal of the process (if any) in the same format used by the *who* utility.

comm The name of the command being executed (*argv*[0] value) as a string.

args The command with all its arguments as a string. The implementation may truncate this value to the field width; it is implementation-dependent whether any further truncation occurs. It is unspecified whether the string represented is a version of the argument list as it was passed to the command when it started, or is a version of the arguments as they may have been modified by the application. Applications cannot depend on being able to modify their argument list and having that modification be reflected in the output of *ps*.

Any field need not be meaningful in all implementations. In such a case a hyphen (–) should be output in place of the field value.

Only **comm** and **args** are allowed to contain blank characters; all others are not. Any implementation-dependent variables will be specified in the system documentation along with the default header and indicating if the field may contain blank characters.

The following table specifies the default header to be used in the POSIX locale corresponding to each format specifier.

Format Specifier	Default Header	Format Specifier	Default Header
args	COMMAND	ppid	PPID
comm	COMMAND	rgroup	RGROUP
etime	ELAPSED	ruser	RUSER
group	GROUP	time	TIME
nice	NI	tty	TT
pcpu	%CPU	user	USER
pgid	PGID	vsz	VSZ
pid	PID		

Table 3-14 Variable Names and Default Headers in *ps*

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

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

The **args** format specifier is allowed to produce a truncated version of the command arguments. In some implementations, this information is no longer available when the *ps* utility is executed.

If the field width is too narrow to display a textual ID, the system may use a numeric version. Normally, the system would be expected to choose large enough field widths, but if a large number of fields were selected to write, it might squeeze fields to their minimum sizes to fit on one line. One way to ensure adequate width for the textual IDs is to override the default header for a field to make it larger than most or all user or group names.

There is no special quoting mechanism for header text. The header text is the rest of the argument. If multiple header changes are needed, multiple **–o** options can be used, such as:

```
ps -o "user=User Name" -o pid=Process\ ID
```

On some systems, especially multi-level secure systems, *ps* may be severely restricted and produce information only about child processes owned by the user.

EXAMPLES

The command:

```
ps -o user,pid,ppid=MOM -o args
```

writes the following in the POSIX locale:

```
   USER     PID     MOM     COMMAND
helene       34      12     ps -o uid,pid,ppid=MOM -o args
```

The contents of the **COMMAND** field need not be the same in all implementations, due to possible truncation.

FUTURE DIRECTIONS

None.

SEE ALSO

kill, nice, renice.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

pwd – return working directory name

SYNOPSIS

```
pwd
```

DESCRIPTION

The *pwd* utility will write an absolute pathname of the current working directory to standard output.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *pwd*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The *pwd* utility output will be an absolute pathname of the current working directory:

```
"%s\n", <directory pathname>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If an error is detected, output will not be written to standard output, a diagnostic message will be written to standard error, and the exit status will not be zero.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cd, the **XSH** specification description of *getcwd*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

read – read a line from standard input

SYNOPSIS

`read [-r] var...`

DESCRIPTION

The *read* utility will read a single line from standard input.

By default, unless the –**r** option is specified, backslash (\) acts as an escape character, as described in Section 2.2.1 on page 20. If standard input is a terminal device and the invoking shell is interactive, *read* will prompt for a continuation line when:

- The shell reads an input line ending with a backslash, unless the –**r** option is specified.
- A here-document is not terminated after a newline character is entered.

The line will be split into fields as in the shell (see Section 2.6.5 on page 38); the first field will be assigned to the first variable *var*, the second field to the second variable *var*, and so forth. If there are fewer *var* operands specified than there are fields, the leftover fields and their intervening separators will be assigned to the last *var*. If there are fewer fields than *var*s, the remaining *var*s will be set to empty strings.

The setting of variables specified by the *var* operands will affect the current shell execution environment; see Section 2.12 on page 63. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(read foo)
nohup read ...
find . -exec read ... \;
```

it will not affect the shell variables in the caller's environment.

OPTIONS

The *read* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–**r** Do not treat a backslash character in any special way. Consider each backslash to be part of the input line.

OPERANDS

The following operands are supported:

var The name of an existing or non-existing shell variable.

STDIN

The standard input must be a text file.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *read*:

IFS Determine the internal field separators used to delimit fields. See Section 2.5.3 on page 29.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PS2 Provide the prompt string that an interactive shell will write to standard error when a line ending with a backslash is read and the **–r** option was not specified, or if a here-document is not terminated after a newline character is entered.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used for diagnostic messages and prompts for continued input.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 End-of-file was detected or an error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The *read* utility has historically been a shell built-in.

The **–r** option is included to enable *read* to subsume the purpose of the *line* utility, which has been marked **TO BE WITHDRAWN**.

The results are undefined if an end-of-file is detected following a backslash at the end of a line when **–r** is not specified.

EXAMPLES
The following command:

```
while read -r xx yy
do
        printf "%s %s\n" "$yy" "$xx"
done < input_file
```

prints a file with the first field of each line moved to the end of the line.

FUTURE DIRECTIONS

None.

SEE ALSO

line.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Relocated from the *sh* description for alignment with the ISO / IEC 9945-2: 1993 standard.

NAME

red – restricted text editor (**WITHDRAWN**)

SYNOPSIS

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

APPLICATION USAGE

This utility has been withdrawn because it does not provide the secure environment it implies.

SEE ALSO

sed, *sh*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an international environment has been described.

Issue 4

This page has been marked **WITHDRAWN**.

NAME

renice – set system scheduling priorities of running processes

SYNOPSIS

```
renice [-n increment][ -g| -p| -u] ID...
```

OB
```
renice nice_value[-p] pid...[-g gid...][-p pid...][-u user]
```

OB
```
renice nice_value -g gid...[-g gid...][-p pid...][-u user]
```

OB
```
renice nice_value -u user...[-g gid...][-p pid...][-u user]
```

DESCRIPTION

The *renice* utility requests that the system scheduling priorities (see the definition of **system scheduling priority** in the **XBD** specification, **Chapter 2**, **Glossary**) of one or more running processes be changed. By default, the applicable processes are specified by their process IDs. When a process group is specified (see **–g**), the request applies to all processes in the process group.

The system scheduling priority is bounded in an implementation-dependent manner. If the
OB requested *increment* (or *nice_value* in the obsolescent versions) would raise or lower the system scheduling priority of the executed utility beyond implementation-dependent limits, then the limit whose value was exceeded is used.

FIPS When a user is *renice*d, the request applies to all processes whose saved set-user-ID matches the user ID corresponding to the user.

Regardless of which options are supplied or any other factor, *renice* will not alter the system scheduling priorities of any process unless the user requesting such a change has appropriate privileges to do so for the specified process. If the user lacks appropriate privileges to perform the requested action, the utility will return an error status.

FIPS The saved set-user-ID of the user's process will be checked instead of its effective user ID when *renice* attempts to determine the user ID of the process in order to determine whether the user has appropriate privileges.

OPTIONS

OB The *renice* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The obsolescent version conforms with the following exceptions:

- The first operand, *nice_value*, must precede the options and can have the appearance of a multi-digit option.
- The **–g**, **–p** and **–u** options can each take multiple option-arguments.
- The *pid* option-argument can be used without its **–p** option.

The following options are supported:

OB **–g** Interpret all operands (or just the *gid* arguments in the obsolescent version) as unsigned decimal integer process group IDs.

–n *increment*
Specify how the system scheduling priority of the specified process or processes is to be adjusted. The *increment* option-argument is a positive or negative decimal integer that will be used to modify the system scheduling priority of the specified process or processes.

Positive *increment* values cause a lower system scheduling priority. Negative *increment* values may require appropriate privileges and will cause a higher system scheduling priority.

OB **–p** Interpret all operands (or just the *pid* arguments in the obsolescent version) as unsigned decimal integer process IDs. The **–p** option is the default if no options are specified.

OB **–u** Interpret all operands (or just the *user* arguments in the obsolescent version) as users. If a user exists with a user name equal to the operand, then the user ID of that user will be used in further processing. Otherwise, if the operand represents an unsigned decimal integer, it will be used as the numeric user ID of the user.

OPERANDS

The following operands are supported:

ID A process ID, process group ID or user name/user ID, depending on the option selected.

OB *nice_value*
The value specified is taken as the actual system scheduling priority, rather than as an increment to the existing system scheduling priority. Specifying a scheduling priority higher than that of the existing process may require appropriate privileges.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *renice*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

1. Adjust the system scheduling priority so that process IDs 987 and 32 would have a lower scheduling priority:

   ```
   renice -n 5 -p 987 32
   ```

2. Adjust the system scheduling priority so that group IDs 324 and 76 would have a higher scheduling priority, if the user has the appropriate privileges to do so:

   ```
   renice -n -4 -g 324 76
   ```

3. Adjust the system scheduling priority so that numeric user ID 8 and user **sas** would have a lower scheduling priority:

   ```
   renice -n 4 -u 8 sas
   ```

Useful nice values on historical systems include 19 or 20 (the affected processes will run only when nothing else in the system attempts to run), 0 (the base scheduling priority), and any negative number (to make processes run faster).

FUTURE DIRECTIONS

None.

SEE ALSO

nice.

CHANGE HISTORY

First released in Issue 4.

NAME

rm – remove directory entries

SYNOPSIS

```
rm [-fiRr] file...
```

DESCRIPTION

The *rm* utility removes the directory entry specified by each *file* argument.

If either of the files dot or dot-dot are specified as the basename portion of an operand (that is, the final pathname component), *rm* will write a diagnostic message to standard error and do nothing more with such operands.

For each *file* the following steps are taken:

1. If the *file* does not exist:
   a. If the **–f** option is not specified, write a diagnostic message to standard error.
   b. Go on to any remaining *files*.
2. If *file* is of type directory, the following steps are taken:
   a. If neither the **–R** option nor the **–r** option is specified, write a diagnostic message to standard error, do nothing more with *file*, and go on to any remaining files.
   b. If the **–f** option is not specified, and either the permissions of *file* do not permit writing and the standard input is a terminal or the **–i** option is specified, write a prompt to standard error and read a line from the standard input. If the response is not affirmative, do nothing more with the current file and go on to any remaining files.
   c. For each entry contained in *file*, other than dot or dot-dot, the four steps listed here (1-4) are taken with the entry as if it were a *file* operand.
   d. If the **–i** option is specified, write a prompt to standard error and read a line from the standard input. If the response is not affirmative, do nothing more with the current file, and go on to any remaining files.
3. If *file* is not of type directory, the **–f** option is not specified, and either the permissions of *file* do not permit writing and the standard input is a terminal or the **–i** option is specified, write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, do nothing more with the current file and go on to any remaining files.
4. If the current file is a directory, *rm* will perform actions equivalent to the **XSH** specification *rmdir*( ) function called with a pathname of the current file used as the *path* argument. If the current file is not a directory, *rm* will perform actions equivalent to the **XSH** specification *unlink*( ) function called with a pathname of the current file used as the *path* argument.

   If this fails for any reason, *rm* will write a diagnostic message to standard error, do nothing more with the current file, and go on to any remaining files.

The *rm* utility is able to descend to arbitrary depths in a file hierarchy, and will not fail due to path length limitations (unless an operand specified by the user exceeds system limitations).

OPTIONS

The *rm* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–f Do not prompt for confirmation. Do not write diagnostic messages or modify the exit status in the case of non-existent operands. Any previous occurrences of the **–i** option will be ignored.

–i Prompt for confirmation as described previously. Any previous occurrences of the **–f** option will be ignored.

–R Remove file hierarchies. See **DESCRIPTION**.

–r Equivalent to **–R**.

OPERANDS

The following operand is supported:

file A pathname of a directory entry to be removed.

STDIN

Used to read an input line in response to each prompt specified in **STDOUT**. Otherwise, the standard input will not be used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *rm*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes, and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments) and the behaviour of character classes within regular expressions used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES

Determine the locale for the processing of affirmative responses that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Prompts are written to standard error under the conditions specified in **DESCRIPTION** and **OPTIONS**. The prompts will contain the *file* pathname, but their format is otherwise unspecified. The standard error is also used for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 If the **–f** option was not specified, all the named directory entries were removed; otherwise, all the existing named directory entries were removed.

>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

It is forbidden to remove the names dot and dot-dot in order to avoid the consequences of inadvertently doing something like:

```
rm –r .*
```

Systems do not permit the removal of the last link to an executable binary file that is being executed; see the [EBUSY] error in the **XSH** specification *unlink*( ) description. Thus, the *rm* utility can fail to remove such files.

The **–i** option causes *rm* to prompt and read the standard input even if the standard input is not a terminal, but in the absence of **–i** the mode prompting is not done when the standard input is not a terminal.

EXAMPLES

1. The following command:

   ```
   rm a.out core
   ```

   removes the directory entries: **a.out** and **core**.

2. The following command:

   ```
   rm -Rf junk
   ```

   removes the directory **junk** and all its contents, without prompting.

FUTURE DIRECTIONS

None.

SEE ALSO

rmdir, the **XSH** specification description of *remove*( ), *unlink*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of these utilities in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

rmdel – remove a delta from an SCCS file (**DEVELOPMENT**)

SYNOPSIS

EX `rmdel -r SID file...`

DESCRIPTION

The *rmdel* utility removes the delta specified by the SID from each named SCCS file. The delta to be removed must be the 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; that is, if a *p-file* (see *get*) exists for the named SCCS file, the SID specified must not appear in any entry of the *p-file*.

Removal of a delta is restricted to: (1) the user who made the delta; (2) the owner of the SCCS file; (3) the owner of the directory containing the SCCS file.

OPTIONS

The *rmdel* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–r *SID* Specify the SCCS identification string (*SID*) of the delta to be deleted.

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *rmdel* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with **s.**) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only when the *file* operand is specified as –. Each line of the text file is interpreted as an SCCS pathname.

INPUT FILES

The SCCS files are files of unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *rmdel*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The SCCS files are files of unspecified format. During processing of a *file*, a temporary *x-file*, as described in *admin*, may be created and deleted; a locking *z-file*, as described in *get*, may be created and deleted.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
delta, *get*, *prs*.

CHANGE HISTORY
First released in Issue 2.

Issue 3
Functionally equivalent to the entry in Issue 2.

Issue 4
Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

rmdir – remove directories

SYNOPSIS

`rmdir [-p] dir...`

DESCRIPTION

The *rmdir* utility will remove the directory entry specified by each *dir* operand, which must refer to an empty directory.

Directories will be processed in the order specified. If a directory and a subdirectory of that directory are specified in a single invocation of the *rmdir* utility, the subdirectory must be specified before the parent directory so that the parent directory will be empty when the *rmdir* utility tries to remove it.

OPTIONS

The *rmdir* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–p Remove all directories in a pathname. For each *dir* operand:

1. The directory entry it names will be removed.
2. If the *dir* operand includes more than one pathname component, effects equivalent to the following command will occur:

```
rmdir –p $(dirname dir)
```

OPERANDS

The following operand is supported:

dir A pathname of an empty directory to be removed.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *rmdir*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Each directory entry specified by a *dir* operand was removed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The definition of an empty directory is one that contains, at most, directory entries for dot and dot-dot.

EXAMPLES
If a directory **a** in the current directory is empty except it contains a directory **b** and **a/b** is empty except it contains a directory **c**:

```
rmdir -p a/b/c
```

will remove all three directories.

FUTURE DIRECTIONS
None.

SEE ALSO
rm, the **XSH** specification description of *remove*( ), *rmdir*( ), *unlink*( ).

CHANGE HISTORY
First released in Issue 2.

Issue 3
The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4
Separated from the *rm* description and aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

sact – print current SCCS file-editing activity (**DEVELOPMENT**)

SYNOPSIS

EX `sact` *`file...`*

DESCRIPTION

The *sact* utility informs the user of any impending deltas to a named SCCS file by writing a list to standard output. This situation occurs when *get* –**e** has been executed previously without a subsequent execution of *delta*.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *sact* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only when the *file* operand is specified as –. Each line of the text file is interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files interrogated are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sact*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The output for each named file consists of a line in the following format:

"%s␣%s␣%s␣%s␣%s\n", <*SID*>, <*new SID*>, <*login*>, <*date*>, <*time*>

<*SID*> Specifies the SID of a delta that currently exists in the SCCS file to which changes will be made to make the new delta.

<*new SID*>
Specifies the SID for the new delta to be created.

<*login*> Contains the login name of the user who will make the delta (that is, who executed a *get* for editing).

<*date*> Contains the date that *get* **−e** was executed, in the format used by the *prs* **:D:** data keyword.

<*time*> Contains the time that *get* **−e** was executed, in the format used by the *prs* **:T:** data keyword.

If there is more than one named file or if a directory or standard input is named, each pathname is written before each of the preceding lines:

"\n%s:\n", <*pathname*>

STDERR

Used only for optional informative messages concerning SCCS files with no impending deltas, and for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

delta, *get*, *unget*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

Issue 4, Version 2

The **STDERR** section encompasses informative messages concerning SCCS files with no impending deltas.

NAME

sccs – front end for the SCCS subsystem (**DEVELOPMENT**)

SYNOPSIS

EX `sccs [-r][-d path][-p path] command [options...][operands...]`

DESCRIPTION

The *sccs* utility is a front end to the SCCS programs. It also includes the capability to run set-user-id to another user to provide additional protection.

The *sccs* utility invokes the specified *command* with the specified *options* and *operands*. By default, each of the *operands* is modified by prefixing it with the string SCCS/s..

The *command* operand can be one of the SCCS utilities in this document (*admin*, *delta*, *get*, *prs*, *rmdel*, *sact*, *unget*, *val* or *what*) or one of the pseudo-utilities listed in **EXTENDED DESCRIPTION**.

OPTIONS

The *sccs* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that *options* operands are actually options to be passed to the utility named by *command*. When the portion of the command:

```
command [options ...] [operands ...]
```

is considered, all of the pseudo-utilities used as *command* support the Utility Syntax Guidelines. Any of the other SCCS utilities that can be invoked in this manner support the Guidelines to the extent indicated by their individual **OPTIONS** sections.

The following options are supported preceding the *command* operand:

–**d** *path* A pathname of a directory to be used as a root directory for the SCCS files. The default is the current directory. The –**d** option takes precedence over the *PROJECTDIR* variable. See –**p**.

–**p** *path* A pathname of a directory in which the SCCS files are located. The default is the **SCCS** directory.

The –**p** option differs from the –**d** option in that the –**d** option-argument is prefixed to the entire pathname and the –**p** option-argument is inserted before the final component of the pathname. For example:

```
sccs -d /x -p y get a/b
```

will convert to:

```
get /x/a/y/s.b
```

This allows the creation of aliases such as:

```
alias syssccs="sccs -d /usr/src"
```

which will be used as:

```
syssccs get cmd/who.c
```

–**r** Invoke *command* with the real user ID of the process, not any effective user ID that the *sccs* utility is set to. Certain commands (*admin*, **check**, **clean**, **diffs**, **info**, *rmdel* and **tell**) cannot be run set-user-ID by all users, since this would allow anyone to change the authorisations. These commands are always run as the real user.

OPERANDS

The following operands are supported:

command
: An SCCS utility name or the name of one of the pseudo-utilities listed in **EXTENDED DESCRIPTION**.

options An option or option-argument to be passed to *command*.

operands
: An operand to be passed to *command*.

STDIN

See the utility description for the specified *command*.

INPUT FILES

See the utility description for the specified *command*.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sccs*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PROJECTDIR
: Provide a default value for the **−d** *path* option. If the value of *PROJECTDIR* begins with a slash, it is considered an absolute pathname; otherwise, the home directory of a user of that name is examined for a subdirectory **src** or **source**. If such a directory is found, it is used. Otherwise, the value is used as a relative pathname.

Additional environment variable effects may be found in the utility description for the specified *command*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

See the utility description for the specified *command*.

STDERR

See the utility description for the specified *command*.

OUTPUT FILES

See the utility description for the specified *command*.

EXTENDED DESCRIPTION

The following pseudo-utilities are supported as *command* operands. All options referred to in the following list are values given in the *options* operands following *command*.

check Equivalent to **info**, except that nothing is printed if nothing is being edited, and a non-zero exit status is returned if anything is being edited. The intent is to have this included in an ‘‘install’’ entry in a makefile to ensure that everything is included into the SCCS file before a version is installed.

clean Remove everything from the current directory that can be recreated from SCCS files, but do not remove any files being edited. If the **–b** option is given, branches are ignored in the determination of whether they are being edited; this is dangerous if branches are kept in the same directory.

create Create an SCCS file, taking the initial contents from the file of the same name. Any options to *admin* are accepted. If the creation is successful, the original files are renamed by prefixing the basenames with a comma. These renamed files should be removed after it has been verified that the SCCS files have been created successfully.

delget Perform a *delta* on the named files and then *get* new versions. The new versions will have ID keywords expanded and will not be editable. Any **–m**, **–p**, **–r**, **–s**, and **–y** options will be passed to *delta*, and any **–b**, **–c**, **–e**, **–i**, **–k**, **–l**, **–s** and **–x** options will be passed to *get*.

deledit Equivalent to **delget**, except that the *get* phase includes the **–e** option. This option is useful for making a checkpoint of the current editing phase. The same options will be passed to *delta* as described above, and all the options listed for *get* above except **–e** are passed to **edit**.

diffs Write a difference listing between the current version of the files checked out for editing and the versions in SCCS format. Any **–r**, **–c**, **–i**, **–x** and **–t** options are passed to *get*; any **–l**, **–s**, **–e**, **–f**, **–h** and **–b** options are passed to *diff*. A **–C** option is passed to *diff* as **–c**.

edit Equivalent to *get* **–e**.

fix Remove the named delta, but leave a copy of the delta with the changes that were in it. It is useful for fixing small compiler bugs, and so forth. It must be followed by a **–r** *SID* option. Since **fix** doesn’t leave audit trails, it should be used carefully.

info Write a listing of all files being edited. If the **–b** option is given, branches (that is, SIDs with two or fewer components) are ignored. If a **–u** *user* option is given, then only files being edited by the named user are listed. A **–U** option is equivalent to **–u** <*current user*>.

print Write out verbose information about the named files, equivalent to *sccs prs*.

tell Write a newline-separated list of the files being edited to standard output. Takes the **–b**, **–u** and **–U** options like **info** and **check**.

unedit This is the opposite of an **edit** or a *get* **–e**. It should be used with caution, since any changes made since the *get* will be lost.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Many of the SCCS utilities take directory names as operands as well as specific filenames. The pseudo-utilities supported by *sccs* are not described as having this capability, but are not prohibited from doing so.

EXAMPLES

1. To get a file for editing, edit it and produce a new delta:

```
sccs get -e file.c
ex file.c
sccs delta file.c
```

2. To get a file from another directory:

```
sccs -p /usr/src/sccs/s. get cc.c
```

or:

```
sccs get /usr/src/sccs/s.cc.c
```

3. To make a delta of a large number of files in the current directory:

```
sccs delta *.c
```

4. To get a list of files being edited that are not on branches:

```
sccs info -b
```

5. To delta everything being edited by the current user:

```
sccs delta $(sccs tell -U)
```

6. In a makefile, to get source files from an SCCS file if it does not already exist:

```
SRCS = <list of source files>
$(SRCS):
        sccs get $(REL) $@
```

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *delta*, *get*, *make*, *prs*, *rmdel*, *sact*, *unget*, *val*, *what*.

CHANGE HISTORY

First released in Issue 4.

NAME

sdb – symbolic debugger (**WITHDRAWN**)

SYNOPSIS

OP UN `sdb [`*`objfile`* `[`*`corfile`* `[`*`directory-list`*`]]]`

APPLICATION USAGE

This utility has been withdrawn because, by their very nature, symbolic debuggers are strongly system-dependent. Although *sdb* has been withdrawn from this issue, strong efforts will be made to provide it, or similar functionality. Application developers should look at the documentation for their current system; common alternatives are *adb*, *dbx* and *cdb*.

SEE ALSO

cc, *c89*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

This page has been marked **WITHDRAWN**.

NAME

sed – stream editor

SYNOPSIS

```
sed [-n] script[file...]
sed [-n][-e script]...[-f script_file]...[file...]
```

DESCRIPTION

The *sed* utility is a stream editor that reads one or more text files, makes editing changes according to a script of editing commands, and writes the results to standard output. The script is obtained from either the *script* operand string or a combination of the option-arguments from the **–e** *script* and **–f** *script_file* options.

OPTIONS

The *sed* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the order of presentation of the **–e** and **–f** options is significant.

The following options are supported:

–e *script*
: Add the editing commands specified by the *script* option-argument to the end of the script of editing commands. The *script* option-argument has the same properties as the *script* operand, described in **OPERANDS**.

–f *script_file*
: Add the editing commands in the file *script_file* to the end of the script.

–n
: Suppress the default output (in which each line, after it is examined for editing, is written to standard output). Only lines explicitly selected for output will be written.

Multiple **–e** and **–f** options may be specified. All commands are added to the script in the order specified, regardless of their origin.

OPERANDS

The following operands are supported:

file
: A pathname of a file whose contents will be read and edited. If multiple *file* operands are specified, the named files will be read in the order specified and the concatenation will be edited. If no *file* operands are specified, the standard input will be used.

script
: A string to be used as the script of editing commands. The application must not present a *script* that violates the restrictions of a text file except that the final character need not be a newline character.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files must be text files. The *script_file*s named by the **–f** option will consist of editing commands, one per line.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sed*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments and input files), and the behaviour of character classes within regular expressions.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The input files are written to standard output, with the editing commands specified in the script applied. If the **–n** option is specified, only those input lines selected by the script will be written to standard output.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The output files are text files whose formats are dependent on the editing commands given.

EXTENDED DESCRIPTION
The *script* consists of editing commands, one per line, of the following form:

[*address***[,***address***]]***command***[***arguments***]**

Zero or more blank characters are accepted before the first address and before *command*.

In default operation, *sed* cyclically copies a line of input, less its terminating newline character, 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 standard output (except when **–n** is specified) and deletes the pattern space. Whenever the pattern space is written to standard output or a named file, *sed* will immediately follow it with a newline character.

Some of the commands use a *hold space* to save all or part of the *pattern space* for subsequent retrieval. The *pattern* and *hold spaces* will each be able to hold at least 8192 bytes.

Addresses in sed

An address is either empty, a decimal number that counts input lines cumulatively across files, a $ character that addresses the last line of input, or a context address (which consists of a regular expression as described in **Regular Expressions in sed**, preceded and followed by a delimiter, usually a slash).

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 to 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 will be selected.) Starting at the first line following the selected range, *sed* looks again for the first address. Thereafter the process is repeated.

Editing commands can be applied only to non-selected pattern spaces by use of the negation command ! (see **Editing Commands in sed**).

Regular Expressions in sed

The *sed* utility supports the basic regular expressions described in the **XBD** specification, **Section 7.3**, **Basic Regular Expressions**, with the following additions:

- In a context address, the construction *cREc*, where *c* is any character other than a backslash or newline character, is identical to /*RE*/. If the character designated by *c* appears following a backslash, then it is considered to be that literal character, which does not terminate the RE. For example, 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 character embedded in the pattern space. A literal newline character must not be used in the regular expression of a context address or in the substitute command.

Editing Commands in sed

In the following list of commands, the maximum number of permissible addresses for each command is indicated by **[***0addr***]**, **[***1addr***]** or **[***2addr***]**, representing zero, one or two addresses.

The argument *text* consists of one or more lines. Each embedded newline character in the text must be preceded by a backslash. Other backslashes in text are removed and the following character is treated literally.

The **r** and **w** commands take an optional *rfile* (or *wfile*) parameter, separated from the command letter by one or more blank characters; implementations may allow zero separation as an extension.

The argument *rfile* or the argument *wfile* terminates the command line. Each *wfile* will be created
EX before processing begins. Implementations support at least ten *wfile* arguments in the script; the
EX actual number (greater than or equal to 10) that will be supported by the implementation is unspecified. The use of the *wfile* parameter causes that file to be initially created, if it does not exist, or will replace the contents of an existing file.

The **b**, **r**, **s**, **t**, **w**, **y**, **!** and **:** commands accept additional arguments. The following synopses indicate which arguments must be separated from the commands by a single space character.

Two of the commands take a *command-list*, which is a list of *sed* commands separated by newline characters, as follows:

```
{ command
command
...
}
```

The { can be preceded with blank characters and can be followed with white space. The *commands* can be preceded by white space. The terminating } must be preceded by a newline character and then zero or more blank characters.

[*2addr***]** **{***command-list*
} Execute *command-list* only when the pattern space is selected.

[*1addr***]a**
text Write *text* to standard output just before each attempt to fetch a line of input, whether by executing the **N** command or by beginning a new cycle.

[*2addr***]b [***label***]**
Branch to the **:** command bearing the *label*. If *label* is not specified, branch to the end of the script. The implementation supports *labels* recognised as unique up to at least 8 characters; the actual length (greater than or equal to 8) that is supported by the implementation is unspecified. It is unspecified whether exceeding a label length causes an error or a silent truncation.

[*2addr***]c**
text Delete the pattern space. With a 0 or 1 address or at the end of a 2-address range, place *text* on the output.

[*2addr***]d**
Delete the pattern space and start the next cycle.

[*2addr***]D**
Delete the initial segment of the pattern space up to and including the first newline character and start the next cycle.

[*2addr***]g**
Replace the contents of the pattern space by the contents of the hold space.

[*2addr***]G**
Append to the pattern space a newline character followed by the contents of the hold space.

[*2addr***]h**
Replace the contents of the hold space with the contents of the pattern space.

[*2addr***]H**
Append to the hold space a newline character followed by the contents of the pattern space.

[*1addr***]i**
text Write *text* to standard output.

[*2addr***]l**
(The letter ell.) Write the pattern space to standard output in a visually unambiguous form. The characters listed in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \r, \t, \v) will be written as the corresponding escape sequence; the \n in that table is not applicable. Non-printable characters not in that

table will be written as one three-digit octal number (with a preceding backslash character) for each byte in the character (most significant byte first). If the size of a byte on the system is greater than nine bits, the format used for non-printable characters is implementation-dependent.

Long lines will be folded, with the point of folding indicated by writing a backslash followed by a newline character; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line will be marked with a $.

[*2addr***]n**
Write the pattern space to standard output if the default output has not been suppressed, and replace the pattern space with the next line of input.

[*2addr***]N**
Append the next line of input to the pattern space, using an embedded newline character to separate the appended material from the original material. Note that the current line number changes.

[*2addr***]p**
Write the pattern space to standard output.

[*2addr***]P**
Write the pattern space, up to the first newline character, to standard output.

[*1addr***]q**
Branch to the end of the script and quit without starting a new cycle.

[*1addr***]r** *rfile*
Copy the contents of *rfile* to standard output just before each attempt to fetch a line of input. If *rfile* does not exist or cannot be read, it is treated as if it were an empty file, causing no error condition.

[*2addr***]s/***regular expression/replacement/flags*
Substitute the *replacement* string for instances of the *regular expression* in the pattern space. Any character other than backslash or newline can be used instead of a slash to delimit the RE and the replacement. Within the RE and the replacement, the RE delimiter itself can be used as a literal character if it is preceded by a backslash.

An ampersand (**&**) appearing in the *replacement* will be replaced by the string matching the RE. The special meaning of & in this context can be suppressed by preceding it by backslash. The characters *n*, where *n* is a digit, will be replaced by the text matched by the corresponding backreference expression. For each backslash (\) encountered in scanning *replacement* from beginning to end, the following character loses its special meaning (if any). It is unspecified what special meaning is given to any character other than &, \ or digits.

A line can be split by substituting a newline character into it. The application must escape the newline character in the *replacement* by preceding it by backslash. A substitution is considered to have been performed even if the replacement string is identical to the string that it replaces.

The value of *flags* must be zero or more of:

n Substitute for the *n*th occurrence only of the *regular expression* found within the pattern space.

g Globally substitute for all non-overlapping instances of the *regular expression* rather than just the first one. If both g and *n* are specified, the results are

unspecified.

p Write the pattern space to standard output if a replacement was made.

w *wfile* Write. Append the pattern space to *wfile* if a replacement was made.

[*2addr***]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 not specified, branch to the end of the script.

[*2addr***]w** *wfile*
Append (write) the pattern space to *wfile*.

[*2addr***]x**
Exchange the contents of the pattern and hold spaces.

[*2addr***]y/***string1***/***string2***/**
Replace all occurrences of characters in *string1* with the corresponding characters in *string2*. If the number of characters in *string1* and *string2* are not equal, or if any of the characters in *string1* appear more than once, the results are undefined. Any character other than backslash or newline can be used instead of slash to delimit the strings. Within *string1* and *string2*, the delimiter itself can be used as a literal character if it is preceded by a backslash.

[*2addr***]!***command*
[*2addr***]!{***command-list*
} Apply the *command* or *command-list* only to the lines that are not selected by the addresses.

[*0addr***]:***label*
This command does nothing; it bears a *label* for the **b** and **t** commands to branch to.

[*1addr***]=**
Write the following to standard output:

```
"%d\n", <current line number>
```

[*0addr***]** An empty command is ignored.

[*0addr***]#**
The # and the remainder of the line are ignored (treated as a comment), with the single exception that if the first two characters in the file are #n, the default output is suppressed; this is the equivalent of specifying **–n** on the command line.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Regular expressions match entire strings, not just individual lines, but a newline character is matched by \n in a *sed* RE; a newline character is not allowed in an RE. Also note that \n cannot be used to match a newline character at the end of an input line; newline characters appear in the pattern space as a result of the **N** editing command.

EXAMPLES

This *sed* script simulates the BSD *cat* –**s** command, squeezing excess blank lines from standard input.

```
sed -n '
# Write non-empty lines.
/./   {
      p
      d
      }
# Write a single empty line, then look for more empty lines.
/^$/  p
# Get next line, discard the held <newline> (empty line),
# and look for more empty lines.
:Empty
/^$/  {
      N
      s/.//
      b Empty
      }
# Write the non-empty line before going back to search
# for the first in a set of empty lines.
      p
'
```

FUTURE DIRECTIONS

None.

SEE ALSO

awk, *ed*, *grep*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The regular expressions have been marked as being of the internationalised type on systems supporting an internationalised environment.

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

sh – shell, the standard command language interpreter

SYNOPSIS

EX `sh [-abCefimnuvx][-o option][+abCefmnuvx][+o option] [command_file [argument...]]`

EX `sh [-abCefimnuvx][-o option][+abCefmnuvx][+o option]command_string [command_name [argument...]]`

EX `sh -s[-abCefimnuvx][-o option][+abCefmnuvx][+o option][argument...]]`

DESCRIPTION

The *sh* utility is a command language interpreter that executes commands read from a command-line string, the standard input or a specified file. The commands to be executed must be expressed in the language described in Chapter 2 on page 19.

OPTIONS

The *sh* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The **–a**, **–b**, **–C**, **–e**, **–f**, **–m**, **–n**, **–o** *option*, **–u**, **–v** and **–x** options are described as part of the *set*
EX utility in Section 2.14 on page 67. The option letters derived from the *set* special built-in are also accepted with a leading plus sign (+) instead of a leading hyphen (meaning the reverse case of the option as described in this document).

The following additional options are supported:

–c Read commands from the *command_string* operand. Set the value of special parameter 0 (see Section 2.5.2 on page 27) from the value of the *command_name* operand and the positional parameters ($1, $2, and so forth) in sequence from the remaining *argument* operands. No commands will be read from the standard input.

–i Specify that the shell is *interactive*; see below. An implementation may treat specifying the **–i** option as an error if the real user ID of the calling process does not equal the effective user ID or if the real group ID does not equal the effective user ID.

–s Read commands from the standard input.

If there are no operands and the **–c** option is not specified, the **–s** option is assumed.

If the **–i** option is present, or if there are no operands and the shell's standard input and standard error are attached to a terminal, the shell is considered to be *interactive*.

OPERANDS

The following operands are supported:

– A single hyphen is treated as the first operand and then ignored. If both – and –– are given as arguments, or if other operands precede the single hyphen, the results are undefined.

argument

The positional parameters ($1, $2 and so forth) will be set to *arguments*, if any.

command_file

The pathname of a file containing commands. If the pathname contains one or more slash characters, the implementation will attempt to read that file; the file need not be executable. If the pathname does not contain a slash character:

- The implementation will attempt to read that file from the current working directory; the file need not be executable.

- If the file is not in the current working directory, the implementation may perform a search for an executable file using the value of *PATH*, as described in **Command Search and Execution** on page 47.

Special parameter 0 (see Section 2.5.2 on page 27) is set to the value of *command_file*. If *sh* is called using a synopsis form that omits *command_file*, special parameter 0 is set to the value of the first argument passed to *sh* from its parent (for example, *argv*[0] for a C program), which is normally a pathname used to execute the *sh* utility.

command_name

A string assigned to special parameter 0 when executing the commands in *command_string*. If *command_name* is not specified, special parameter 0 is set to the value of the first argument passed to *sh* from its parent (for example, *argv*[0] for a C program), which is normally a pathname used to execute the *sh* utility.

command_string

A string that is interpreted by the shell as one or more commands, as if the string were the argument to the **XSH** specification *system*( ) function. If the *command_string* operand is an empty string, *sh* will exit with a zero exit status.

STDIN

The standard input will be used only if one of the following is true:

- The **–s** option is specified.
- The **–c** option is not specified and no operands are specified.
- The script executes one or more commands that require input from standard input (such as a *read* command that does not redirect its input).

See **INPUT FILES**.

When the shell is using standard input and it invokes a command that also uses standard input, the shell ensures that the standard input file pointer points directly after the command it has read when the command begins execution. It will not read ahead in such a manner that any characters intended to be read by the invoked command are consumed by the shell (whether interpreted by the shell or not) or that characters that are not read by the invoked command are not seen by the shell. When the command expecting to read standard input is started asynchronously by an interactive shell, it is unspecified whether characters are read by the command or interpreted by the shell.

If the standard input to *sh* is a FIFO or terminal device and is set to non-blocking reads, then *sh* will enable blocking reads on standard input. This will remain in effect when the command completes. (This concerns an instance of *sh* that has been invoked, probably by a C-language program, with standard input that has been opened using the O_NONBLOCK flag; see *open*( ) in the **XSH** specification. If the shell did not reset this flag, it would immediately terminate because no input data would be available yet and that would be considered the same as end-of-file.)

INPUT FILES

The input file must be a text file, except that line lengths are unlimited. If the input file is empty or consists solely of blank lines or comments, or both, *sh* will exit with a zero exit status.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sh*:

FCEDIT

This variable, when expanded by the shell, determines the default value for the −**e** *editor* option's *editor* option-argument. If *FCEDIT* is null or unset, *ed* will be used as the editor.

HISTFILE

Determine a pathname naming a command history file. If the *HISTFILE* variable is not set, the shell may attempt to access or create a file **.sh_history** in the user's home directory. If the shell cannot obtain both read and write access to, or create, the history file, it will use an unspecified mechanism that allows the history to operate properly. (References to history ''file'' in this section are understood to mean this unspecified mechanism in such cases.) An implementation may choose to access this variable only when initialising the history file; this initialisation will occur when *fc* or *sh* first attempt to retrieve entries from, or add entries to, the file, as the result of commands issued by the user, the file named by the *ENV* variable, or implementation-dependent system startup files. (The initialisation process for the history file can be dependent on the system startup files, in that they may contain commands that will effectively preempt the user's settings of *HISTFILE* and *HISTSIZE*. For example, function definition commands are recorded in the history file, unless the *set* −**o** nolog option is set. If the system administrator includes function definitions in some system startup file called before the *ENV* file, the history file will be initialised before the user gets a chance to influence its characteristics.) In some historical shells, the history file is initialised just after the *ENV* file has been processed. Therefore, it is implementation-dependent whether changes made to *HISTFILE* after the history file has been initialised are effective. Implementations may choose to disable the history list mechanism for users with appropriate privileges who do not set *HISTFILE*; the specific circumstances under which this will occur are implementation-dependent. If more than one instance of the shell is using the same history file, it is unspecified how updates to the history file from those shells interact. As entries are deleted from the history file, they will be deleted oldest first. It is unspecified when history file entries are physically removed from the history file.

HISTSIZE

Determine a decimal number representing the limit to the number of previous commands that are accessible. If this variable is unset, an unspecified default greater than or equal to 128 will be used. The maximum number of commands in the history list is unspecified, but will be at least 128. An implementation may choose to access this variable only when initialising the history file, as described under *HISTFILE*. Therefore, it is unspecified whether changes made to *HISTSIZE* after the history file has been initialised are effective.

HOME Determine the pathname of the user's home directory. The contents of *HOME* are used in Tilde Expansion as described in Section 2.6.1 on page 32.

IFS *Input field separators*: a string treated as a list of characters that is used for field splitting and to split lines into words with the *read* command. See Section 2.6.5 on page 38. If *IFS* is not set, the shell behaves as if the value of *IFS* were the space, tab and newline characters. Implementations may ignore the value of *IFS* in the environment at the time *sh* is invoked, treating *IFS* as if it were not set.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific

default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the behaviour of range expressions, equivalence classes and multi-character collating elements within pattern matching.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments and input files), which characters are defined as letters (character class **alpha**), and the behaviour of character classes within pattern matching.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

MAIL Determine a pathname of the user's mailbox file for purposes of incoming mail notification. If this variable is set, the shell will inform the user if the file named by the variable is created or if its modification time has changed. Informing the user is accomplished by writing a string of unspecified format to standard error prior to the writing of the next primary prompt string after the completion of an interval defined by the *MAILCHECK* variable. The user will be informed only if *MAIL* is set and *MAILPATH* is not set.

MAILCHECK
Establish a decimal integer value that specifies how often (in seconds) the shell will check for the arrival of mail in the files specified by the *MAILPATH* or *MAIL* variables. The default value is 600 seconds. If set to zero, the shell will check before issuing each primary prompt.

MAILPATH
Provide a list of pathnames and optional messages separated by colons. If this variable is set, the shell will inform the user if any of the files named by the variable are created or if any of their modification times change. (See the preceding entry for *MAIL* for descriptions of mail arrival and user informing.) Each pathname can be followed by % and a string that will be subjected to parameter expansion and written to standard error when the modification time changes. If a % character in the pathname is preceded by a backslash, it will be treated as a literal % in the pathname. The default message is unspecified.

The *MAILPATH* environment variable takes precedence over the *MAIL* variable.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Establish a string formatted as described in the **XBD** specification, **Chapter 6**, **Environment Variables**, used to effect command interpretation. See **Command Search and Execution** on page 47.

ASYNCHRONOUS EVENTS
Default.

STDOUT

See **STDERR**.

STDERR

Except as otherwise stated (by the descriptions of any invoked utilities or in interactive mode), standard error is used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

See Chapter 2. The following additional capabilities are supported.

Command History List

When the *sh* utility is being used interactively, it maintains a list of commands previously entered from the terminal in the file named by the *HISTFILE* environment variable. The type, size and internal format of this file are unspecified. Multiple *sh* processes can share access to the file for a user, if file access permissions allow this; see the description of the *HISTFILE* environment variable.

Command Line Editing

When *sh* is being used interactively from a terminal, the current command and the command history (see *fc*) can be edited using *vi-mode* command line editing. This mode uses commands, described below, similar to a subset of those described in the *vi* utility. Implementations may offer other command line editing modes corresponding to other editing utilities.

The command *set* **–o v** enables vi-mode editing and places *sh* into *vi* insert mode (see **Command Line Editing (vi-mode)**). This command also disables any other editing mode that the implementation may provide. The command *set* **+o v** disables vi-mode editing.

Certain block-mode terminals may be unable to support shell command line editing. If a terminal is unable to provide either edit mode, it need not be possible to *set* **–o v** when using the shell on this terminal.

In the following sections, the characters erase, interrupt, kill and end-of-file are those set by the *stty* utility.

Command Line Editing (vi-mode)

With vi-mode enabled, *sh* can be switched between insert mode and command mode.

When in insert mode, an entered character will be inserted into the command line, except as noted in **vi Line Editing Insert Mode** on page 636. Upon entering *sh* and after termination of the previous command, *sh* will be in insert mode.

Typing an escape character wll switch *sh* into command mode (see **vi Line Editing Command Mode** on page 636). In command mode, an entered character will either invoke a defined operation, be used as part of a multi-character operation or be treated as an error. A character that is not recognised as part of an editing command will terminate any specific editing command and will alert the terminal. Typing the *interrupt* character in command mode will cause *sh* to terminate command line editing on the current command line, reissue the prompt on the next line of the terminal and reset the command history (see *fc*) so that the most recently executed command is the previous command (that is, the command that was being edited when it was interrupted is not reentered into the history).

In the following sections, the phrase ‘‘move the cursor to the beginning of the word’’ means ‘‘move the cursor to the first character of the current word’’ and the phrase ‘‘move the cursor to the end of the word’’ means ‘‘move the cursor to the last character of the current word’’. The phrase ‘‘beginning of the command line’’ indicates the point between the end of the prompt string issued by the shell (or the beginning of the terminal line, if there is no prompt string) and the first character of the command text.

vi Line Editing Insert Mode

While in insert mode, any character typed will be inserted in the current command line, unless it is from the following set.

newline
: Execute the current command line being edited.

erase
: Delete the character previous to the current cursor position and move the current cursor position back one character. In insert mode, characters will be erased from both the screen and the buffer when backspacing.

interrupt
: Terminate command line editing with the same effects as described for interrupting command mode; see **Command Line Editing (vi-mode)** on page 635.

kill
: Clear all the characters from the input line.

<control>-V
: Insert the next character input, even if the character is otherwise a special insert mode character.

<control>-W
: Delete the characters from the one preceding the cursor to the preceding word boundary. The word boundary in this case is the closer to the cursor of either the beginning of the line or a character that is in neither the **blank** nor **punct** character classification of the current locale.

\
: On some systems, when a backslash is followed by an *erase* or *kill* character, that character will be inserted into the input line. This is not actually a feature of *sh* command line editing insert mode, but one of terminal line drivers when the *stty* **iexten** flag is set. Otherwise, the backslash itself will be inserted into the input line.

end-of-file
: Interpreted as the end of input in *sh*. This interpretation will occur only at the beginning of an input line. If end-of-file is entered other than at the beginning of the line, the results are unspecified.

<ESC>
: Place *sh* into command mode.

vi Line Editing Command Mode

In command mode for the command line editing feature, decimal digits not beginning with 0 that precede a command letter will be remembered. Some commands use these decimal digits as a count number that affects the operation.

The term *motion command* represents one of the commands:

```
<space>  0  b  F  l  W  ^  $  ;  E  f  T  w  |  ,  B  e  h  t
```

Any command that modifies the current line will cause a copy of the current line to be made at the end of the command history, the current line will become that copy, and the edit will be

performed on that copy.

Any command that is preceded by *count* will take a count (the numeric value of any preceding decimal digits). Unless otherwise noted, this count will cause the specified operation to repeat by the number of times specified by the count. Also unless otherwise noted, a *count* that is out of range is considered an error condition and will alert the terminal, but neither the cursor position, nor the command line, will change.

The terms *word* and *bigword* are used as defined in the *vi* description. The term *save buffer* corresponds to the term *unnamed buffer* in *vi*.

The following commands are recognised in command mode:

newline
: Execute the current command line being edited.

<control>-L
: Redraw the current command line. Position the cursor at the same location on the new command line.

#
: Insert the character # at the beginning of the current command line and treat the current command line as a comment. This line will be entered into the command history; see *fc*.

=
: Display the possible shell word expansions (see Section 2.6 on page 31) of the bigword at the current command line position. These expansions will be displayed on subsequent terminal lines. If the bigword contains none of the characters ?, * or [, an asterisk (*) will be implicitly assumed at the end. If any directories are matched, these expansions will have a / character appended. After the expansion, the line will be redrawn, the cursor will be repositioned at the current cursor position, and *sh* will be placed in command mode.

****
: Perform pathname expansion (see Section 2.6.6 on page 39) on the current bigword, up to the largest set of characters that can be matched uniquely. If the bigword contains none of the characters ?, * or [, an asterisk (*) will be implicitly assumed at the end. This maximal expansion then will replace the original bigword in the command line, and the cursor will be placed after this expansion. If the resulting bigword completely and uniquely matches a directory, a / character will be inserted directly after the bigword. If some other file is completely matched, a single space character will be inserted after the bigword. After this operation, *sh* will be placed in insert mode.

*****
: Perform pathname expansion on the current bigword and insert all expansions into the command to replace the current bigword, with each expansion separated by a single space character. If at the end of the line, the current cursor position will be moved to the first column position following the expansions and *sh* will be placed in insert mode. Otherwise, the current cursor position will be the last column position of the first character after the expansions and *sh* will be placed in insert mode. If the current bigword contains none of the characters ?, * or [, before the operation, an asterisk will be implicitly assumed at the end.

@*letter*
: Insert the value of the alias named *_letter*. The symbol *letter* represents a single alphabetic character from the portable character set; implementations may support additional characters as an extension. If the alias *_letter* contains other editing commands, these commands will be performed as part of the insertion. If no alias *_letter* is enabled, this command will have no effect.

[*count***]~**

Convert, if the current character is a lower-case letter, to the equivalent upper-case letter and *vice* versa, as prescribed by the current locale. The current cursor position then will be advanced by one character. If the cursor was positioned on the last character of the line, the case conversion will occur, but the cursor will not advance. If the ~ command is preceded by a *count*, that number of characters will be converted, and the cursor will be advanced to the character position after the last character converted. If the *count* is larger than the number of characters after the cursor, this is not considered an error; the cursor will advance to the last character on the line.

[*count***].**

Repeat the most recent non-motion command, even if it was executed on an earlier command line. If the previous command was preceded by a *count*, and no count is given on the . command, the count from the previous command will be included as part of the repeated command. If the . command is preceded by a *count*, this will override any *count* argument to the previous command. The *count* specified in the . command will become the count for subsequent . commands issued without a count.

[*number***]v**

Invoke the *vi* editor to edit the current command line in a temporary file. When the editor exits, the commands in the temporary file will be executed. If a *number* is prefixed to the command, it specifies the command number in the command history to be edited, rather than the current command line.

[*count***]l** (ell)
[*count***]**<space>

Move the current cursor position to the next character position. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the *count* is larger than the number of characters after the cursor, this is not considered an error; the cursor will advance to the last character on the line.

[*count***]h**

Move the current cursor position to the *count*th (default 1) previous character position. If the cursor was positioned on the first character of the line, the terminal will be alerted and the cursor will not be moved. If the count is larger than the number of characters before the cursor, this is not considered an error; the cursor will move to the first character on the line.

[*count***]w**

Move to the start of the next word. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the *count* is larger than the number of words after the cursor, this is not considered an error; the cursor will advance to the last character on the line.

[*count***]W**

Move to the start of the next bigword. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the *count* is larger than the number of bigwords after the cursor, this is not considered an error; the cursor will advance to the last character on the line.

[*count***]e**

Move to the end of the current word. If at the end of a word, move to the end of the next word. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the *count* is larger than the number of words after the cursor, this is not considered an error; the cursor will

advance to the last character on the line.

[*count***]E**
Move to the end of the current bigword. If at the end of a bigword, move to the end of the next bigword. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the *count* is larger than the number of bigwords after the cursor, this is not considered an error; the cursor will advance to the last character on the line.

[*count***]b**
Move to the beginning of the current word. If at the beginning of a word, move to the beginning of the previous word. If the cursor was positioned on the first character of the line, the terminal will be alerted and the cursor will not be moved. If the *count* is larger than the number of words preceding the cursor, this is not considered an error; the cursor will return to the first character on the line.

[*count***]B**
Move to the beginning of the current bigword. If at the beginning of a bigword, move to the beginning of the previous bigword. If the cursor was positioned on the first character of the line, the terminal will be alerted and the cursor will not be moved. If the *count* is larger than the number of bigwords preceding the cursor, this is not considered an error; the cursor will return to the first character on the line.

^ Move the current cursor position to the first character on the input line that is not a blank character.

$ Move to the last character position on the current command line.

0 (Zero.) Move to the first character position on the current command line.

[*count***] |**
Move to the *count*th character position on the current command line. If no number is specified, move to the first position. The first character position is numbered 1. If the count is larger than the number of characters on the line, this is not considered an error; the cursor will be placed on the last character on the line.

[*count***]f***c*
Move to the first occurrence of the character *c* that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the character *c* does not occur in the line after the current cursor position, the terminal will be alerted and the cursor will not be moved.

[*count***]F***c*
Move to the first occurrence of the character *c* that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal will be alerted and the cursor will not be moved. If the character *c* does not occur in the line before the current cursor position, the terminal will be alerted and the cursor will not be moved.

[*count***]t***c*
Move to the character before the first occurrence of the character *c* that occurs after the current cursor position. If the cursor was positioned on the last character of the line, the terminal will be alerted and the cursor will not be advanced. If the character *c* does not occur in the line after the current cursor position, the terminal will be alerted and the cursor will not be moved.

[*count***]T***c*

Move to the character after the first occurrence of the character *c* that occurs before the current cursor position. If the cursor was positioned on the first character of the line, the terminal will be alerted and the cursor will not be moved. If the character *c* does not occur in the line before the current cursor position, the terminal will be alerted and the cursor will not be moved.

[*count***];**

Repeat the most recent **f**, **F**, **t** or **T** command. Any number argument on that previous command will be ignored. Errors are those described for the repeated command.

[*count***],**

Repeat the most recent f, F, t or T command. Any number argument on that previous command will be ignored. However, reverse the direction of that command.

a Enter insert mode after the current cursor position. Characters that are entered will be inserted before the next character.

A Enter insert mode after the end of the current command line.

i Enter insert mode at the current cursor position. Characters that are entered will be inserted before the current character.

I Enter insert mode at the beginning of the current command line.

R Enter insert mode, replacing characters from the command line beginning at the current cursor position.

[*count***]c** *motion*

Delete the characters between the current cursor position and the cursor position that would result from the specified *motion* command. Then enter insert mode before the first character following any deleted characters. If *count* is specified, it will be applied to the motion command. A *count* will be ignored for the following motion commands:

```
0    ^    $    c
```

If the *motion* command is the character **c**, the current command line will be cleared and insert mode will be entered. If the *motion* command would move the current cursor position toward the beginning of the command line, the character under the current cursor position will not be deleted. If the motion command would move the current cursor position toward the end of the command line, the character under the current cursor position will be deleted. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this is not considered an error; all of the remaining characters in the aforementioned range will be deleted and insert mode will be entered. If the motion command is invalid, the terminal will be alerted, the cursor will not be moved, and no text will be deleted.

C Delete from the current character to the end of the line and enter insert mode at the new end of line.

S Clear the entire current command line and enter insert mode.

[*count***]r***c*

Replace the current character with the character *c*. With a number *count*, replace the current and the following *count*–1 characters. After this command, the current cursor position will be on the last character that was changed. If the *count* is larger than the number of characters after the cursor, this is not considered an error; all of the remaining characters will be changed.

[*count***]_**
Append a space character after the current character position and then append the last bigword in the previous input line after the space character. Then enter insert mode after the last character just appended. With a number *count*, append the *count*th bigword in the previous line.

[*count***]x**
Delete the character at the current cursor position and place the deleted characters in the save buffer. If the cursor was positioned on the last character of the line, the character will be deleted and the cursor position will be moved to the previous character (the new last character). If the *count* is larger than the number of characters after the cursor, this is not considered an error; all the characters from the cursor to the end of the line will be deleted.

[*count***]X**
Delete the character before the current cursor position and place the deleted characters in the save buffer. The character under the current cursor position will not change. If the cursor was positioned on the first character of the line, the terminal will be alerted, and the **X** command will have no effect. If the line contained a single character, the **X** command will have no effect. If the line contained no characters, the terminal will be alerted and the cursor will not be moved. If the *count* is larger than the number of characters before the cursor, this is not considered an error; all the characters from before the cursor to the beginning of the line will be deleted.

[*count***]d** *motion*
Delete the characters between the current cursor position and the character position that would result from the *motion* command. A number *count* repeats the *motion* command *count* times. If the motion command would move toward the beginning of the command line, the character under the current cursor position will not be deleted. If the motion command is **d**, the entire current command line will be cleared. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this is not considered an error; all of the remaining characters in the aforementioned range will be deleted. The deleted characters will be placed in the save buffer.

D Delete all characters from the current cursor position to the end of the line. The deleted characters will be placed in the save buffer.

[*count***]y** *motion*
Yank (that is, copy) the characters from the current cursor position to the position resulting from the *motion* command into a save buffer. A number *count* will be applied to the *motion* command. If the motion command would move toward the beginning of the command line, the character under the current cursor position will not be included in the set of yanked characters. If the motion command is **y**, the entire current command line will be yanked into the save buffer. The current cursor position will be unchanged. If the *count* is larger than the number of characters between the current cursor position and the end of the command line toward which the motion command would move the cursor, this is not considered an error; all of the remaining characters in the aforementioned range will be yanked.

Y Yank the characters from the current cursor position to the end of the line into the save buffer. The current character position will be unchanged.

[*count***]p**
Put a copy of the current contents of the save buffer after the current cursor position.

The current cursor position will be advanced to the last character put from the save buffer. A *count* indicates how many copies of the save buffer will be put.

[*count***]P**
Put a copy of the current contents of the save buffer before the current cursor position. The current cursor position will be moved to the last character put from the save buffer. A *count* indicates how many copies of the save buffer will be put.

u Undo the last command that modified the text of the current command line.

U Undo all changes made to the current command line since first entering command mode on the line.

[*count***]k**
[*count***]**–
Replace the current command line with the previous command line in the shell command history. The cursor will be positioned on the first character of the new command. A count preceding the command will have the same effect as executing the command *count* times. If a **k** or – command retreats past the maximum number of commands in effect for this shell (affected by the *HISTSIZE* environment variable), the terminal will be alerted and the command will have no effect.

[*count***]j**
[*count***]+**
Replace the current command line with the next command line in the shell command history. The cursor will be positioned on the first character of the new command. The command history position will be remembered, and any **k** or – command, or **j** or + command, will decrement or increment that position and then will fetch the line at the new position. If a **j** or command advances past the most recent line in the history, the current command line will be restored to the contents before the first **k** or –.

[*number***]G**
Replace the current command line with the contents of the oldest command line stored in the shell command history. With a number *number*, replace the current command line with the contents of command *number* in the history.

/*string*<newline>
Move backward through the command history, searching for the specified *string*, beginning with the previous command line. If it is not found, the current command line will be unchanged. If it is found in a previous line, this command will behave equivalently to a set of **k** commands to reach that line. If *string* begins with ^, the characters after the ^ will be matched only at the beginning of a line.

?*string*<newline>
Move forward through the command history, searching for the specified string. If it is not found, the current command line will be unchanged. If the string is found in the current command line, the current cursor position will be moved to the beginning of that string. If it is found in the history, this command will behave equivalently to a set of **j** commands to reach that line. If *string* begins with ^, the characters after the ^ will be matched only at the beginning of a line.

n Repeat the most recent **/** or **?** command.

N Repeat the most recent **/** or **?** command, reversing the direction of the search.

EXIT STATUS

The following exit values are returned:

0 The script to be executed consisted solely of zero or more blank lines or comments, or both.
1–125 A non-interactive shell detected a syntax, redirection or variable assignment error.
127 A specified *command_file* could not be found by a non-interactive shell.

Otherwise, the shell will return the exit status of the last command it invoked or attempted to invoke (see also the *exit* utility in Section 2.14 on page 67).

CONSEQUENCES OF ERRORS

See Section 2.8.1 on page 44.

APPLICATION USAGE

Standard input and standard error are the files that determine whether a shell is interactive when **–i** is not specified. For example:

```
sh > file
```

and:

```
sh 2> file
```

create interactive and non-interactive shells, respectively. Although both accept terminal input, the results of error conditions are different, as described in Section 2.8.1 on page 44; in the second example a redirection error encountered by a special built-in utility will abort the shell.

On systems that support set-user-ID scripts, a historical trapdoor has been to link a script to the name **–i**. When it is called by a sequence such as:

```
sh -
```

or by:

```
#! /bin/sh –
```

the historical systems have assumed that no option letters follow. Thus, this document allows the single hyphen to mark the end of the options, in addition to the use of the regular −− argument, because the older practice is so pervasive.

A portable application must protect its first operand, if it starts with a plus sign, by preceding it with the −− argument that denotes the end of the options.

EXAMPLES

1. Execute a shell command from a string:

   ```
   sh -c "cat myfile"
   ```

2. Execute a shell script from a file in the current directory:

   ```
   sh my_shell_cmds
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

cd, *echo*, *pwd*, *test*, *umask*, the **XSH** specification description of *dup*( ), *exec*, *exit*( ), *fork*( ), *pipe*( ), *signal*( ), *system*( ), *ulimit*( ), *umask*( ), *wait*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO/IEC 9945-2:1993 standard. Description of the shell command language and special built-ins moved to Chapter 2 on page 19.

NAME

sleep – suspend execution for an interval

SYNOPSIS

`sleep` *`time`*

DESCRIPTION

The *sleep* utility will suspend execution for at least the integral number of seconds specified by the *time* operand.

OPTIONS

None.

OPERANDS

The following operands are supported:

time A non-negative decimal integer specifying the number of seconds for which to suspend execution.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sleep*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If the *sleep* utility receives a SIGALRM signal, one of the following actions will be taken:

1. Terminate normally with a zero exit status.
2. Effectively ignore the signal.
3. Provide the default behaviour for signals described in the **ASYNCHRONOUS EVENTS** section of the **XBD** specification, **Section 2.1**, **Utility Description Defaults**. This could include terminating with a non-zero exit status.

The *sleep* utility will take the standard action for all other signals.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The execution was successfully suspended for at least *time* seconds, or a SIGALRM signal was received (see **ASYNCHRONOUS EVENTS**).
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
The *sleep* utility can be 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
```

FUTURE DIRECTIONS
None.

SEE ALSO
wait, the **XSH** specification description of *alarm*( ), *sleep*( ).

CHANGE HISTORY
First released in Issue 2.

Issue 3
The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4
Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

sort – sort, merge or sequence check text files

SYNOPSIS

UN `sort [-m][-o output][-bdfinru][-t char][-k keydef]...[-z recsz] [file...]`

UN `sort -c [-bdfinru[-t char][-k keydef...[-z recsz][file...]`

OB UN `sort [-mu][-o output][-bdfir][-t char][+pos1[-pos2]]...[-z recsz] [file...]`

OB UN `sort -c[-u][-bdfinr][-t char][+pos1[-pos2]]...[-z recsz][file]`

DESCRIPTION

The *sort* utility performs one of the following functions:

1. Sorts lines of all the named files together and writes the result to the specified output.
2. Merges lines of all the named (presorted) files together and writes the result to the specified output.
3. Checks that a single input file is correctly presorted.

Comparisons are based on one or more sort keys extracted from each line of input (or the entire line if no sort keys are specified), and are performed using the collating sequence of the current locale.

OPTIONS

OB The *sort* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the notation *+pos1 –pos2* uses a non-standard prefix and multi-digit option names in the obsolescent versions, the **–o** *output* option is recognised after a *file* operand as an obsolescent feature in both versions where the **–c** option is not specified, and the **–k** *keydef* option should follow the **–b**, **–d**, **–f**, **–i**, **–n** and **–r** options.

The following options are supported:

–c Check that the single input file is ordered as specified by the arguments and the collating sequence of the current locale. No output is produced; only the exit code is affected.

–m Merge only; the input file is assumed to be already sorted.

–o *output*
Specify the name of an output file to be used instead of the standard output. This file can be the same as one of the input *file*s.

–u Unique: suppress all but one in each set of lines having equal keys. If used with the **–c** option, check that there are no lines with duplicate keys, in addition to checking that the input file is sorted.

UN **–z** *recsz*
The size of the longest line read in the sort phase is recorded so that buffers of the correct size can be allocated during the merge phase. If the sort phase is omitted via the **–c** or **–m** options, a system-dependent 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. When ordering options appear independent of any key field specifications, the requested field ordering rules are applied

globally to all sort keys. When attached to a specific key (see **–k**), the specified ordering options override all global ordering options for that key. In the obsolescent forms, if one or more of these options follows a *+pos1* option, it will affect only the key field specified by that preceding option.

OB

–d Specify that only blank characters and alphanumeric characters, according to the current setting of LC_CTYPE, are significant in comparisons. The behaviour is undefined for a sort key to which **–i** or **–n** also applies.

–f Consider all lower-case characters that have upper-case equivalents, according to the current setting of LC_CTYPE, to be the upper-case equivalent for the purposes of comparison.

–i Ignore all characters that are non-printable, according to the current setting of LC_CTYPE.

–n Restrict the sort key to an initial numeric string, consisting of optional blank characters, optional minus sign, and zero or more digits with an optional radix character and thousands separators (as defined in the current locale), which will be sorted by arithmetic value. An empty digit string is treated as zero. Leading zeros and signs on zeros do not affect ordering.

–r Reverse the sense of comparisons.

The treatment of field separators can be altered using the options:

–b Ignore leading blank characters when determining the starting and ending positions of a restricted sort key. If the **–b** option is specified before the first **–k** option, it is applied to all **–k** options. Otherwise, the **–b** option can be attached independently to each **–k** *field_start* or *field_end* option-argument (see below).

–t *char*

Use *char* as the field separator character; *char* is not considered to be part of a field (although it can be included in a sort key). Each occurrence of *char* is significant (for example, <*char*><*char*> delimits an empty field). If **–t** is not specified, blank characters are used as default field separators; each maximal non-empty sequence of blank characters that follows a non-blank character is a field separator.

Sort keys can be specified using the options:

–k *keydef*

The *keydef* argument is a restricted sort key field definition. The format of this definition is:

field_start[*type*][,*field_end*[*type*]]

where *field_start* and *field_end* define a key field restricted to a portion of the line (see **EXTENDED DESCRIPTION**), and *type* is a modifier from the list of characters b, d, f, i, n, r. The b modifier behaves like the **–b** option, but applies only to the *field_start* or *field_end* to which it is attached. The other modifiers behave like the corresponding options, but apply only to the key field to which they are attached; they have this effect if specified with *field_start*, *field_end* or both. If any modifier is attached to a *field_start* or to a *field_end*, no option applies to either. Implementations support at least nine occurrences of the **–k** option, which are significant in command line order. If no **–k** option is specified, a default sort key of the entire line is used.

When there are multiple key fields, later keys are compared only after all earlier keys compare equal. Except when the **–u** option is specified, lines that otherwise compare equal are ordered as if none of the options **–d**, **–f**, **–i**, **–n** or **–k** were present (but with **–r**

still in effect, if it was specified) and with all bytes in the lines significant to the comparison. The order in which lines that still compare equal are written is unspecified.

OB *+pos1* Specify the start position of a key field. See **EXTENDED DESCRIPTION**.

OB *–pos2* Specify the end position of a key field. See **EXTENDED DESCRIPTION**.

OPERANDS

The following operand is supported:

file A pathname of a file to be sorted, merged or checked. If no *file* operands are specified, or if a *file* operand is –, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –. See **INPUT FILES**.

INPUT FILES

The input files must be text files, except that the *sort* utility will add a newline character to the end of a file ending with an incomplete last line.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *sort*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for ordering rules.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments and input files) and the behaviour of character classification for the **–b**, **–d**, **–f**, **–i** and **–n** options.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_NUMERIC
Determine the locale for the definition of the radix character and thousands separator for the **–n** option.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Unless the **–o** or **–c** options are in effect, the standard output contains the sorted input.

STDERR

Used for diagnostic messages. A warning message about correcting an incomplete last line of an input file may be generated, but need not affect the final exit status.

OUTPUT FILES

If the **–o** option is in effect, the sorted input is placed in the file *output*.

EXTENDED DESCRIPTION

The notation:

```
–k field_start[type][,field_end[type]]
```

defines a key field that begins at *field_start* and ends at *field_end* inclusive, unless *field_start* falls beyond the end of the line or after *field_end*, in which case the key field is empty. A missing *field_end* means the last character of the line.

A field comprises a maximal sequence of non-separating characters and, in the absence of option **–t**, any preceding field separator.

The *field_start* portion of the *keydef* option-argument has the form:

```
field_number[.first_character]
```

Fields and characters within fields are numbered starting with 1. The *field_number* and *first_character* pieces, interpreted as positive decimal integers, specify the first character to be used as part of a sort key. If *.first_character* is omitted, it refers to the first character of the field.

The *field_end* portion of the *keydef* option-argument has the form:

```
field_number[.last_character]
```

The *field_number* is as described above for *field_start*. The *last_character* piece, interpreted as a non-negative decimal integer, specifies the last character to be used as part of the sort key. If *last_character* evaluates to zero or *.last_character* is omitted, it refers to the last character of the field specified by *field_number*.

If the **–b** option or b type modifier is in effect, characters within a field are counted from the first non-blank character in the field. (This applies separately to *first_character* and *last_character*.)

OB The obsolescent options:

```
[+pos1[-pos2]]
```

provide functionality equivalent to the **–k** *keydef* option. For comparison, the full formats of these options are:

```
+field0_number[.first0_character][type]
     [-field0_number[.first0_character][type] ]
-k field_number[.first_character][type][,field_number[.last_character][type]  ]
```

In the obsolescent form, fields (specified by *field0_number*) and characters within fields (specified by *first0_character*) are numbered from zero instead of one. The optional type modifiers are the same in both forms. If *.first0_character* is omitted or *first0_character* evaluates to zero, it refers to the first character of the field. The **–b** option does not apply to *–pos2*.

The fully specified *+pos1 –pos2* form with type modifiers *T* and *U*:

```
+w.xT -y.zU
```

is equivalent to:

undefined	(*z*==0 & *U* contains b & –t is present)
–k *w*+1.*x*+1*T*,*y*.0*U*	(*z*==0 otherwise)
–k *w*+1.*x*+1*T*,*y*+1.*zU*	(*z* > 0)

As with the non-obsolescent forms, implementations support at least nine occurrences of the *+pos1* option, which are significant in command line order.

EXIT STATUS

The following exit values are returned:

0 All input files were output successfully, or **–c** was specified and the input file was correctly sorted.

1 Under the **–c** option, the file was not ordered as specified, or if the **–c** and **–u** options were both specified, two input lines were found with equal keys.

>1 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The default value for **–t**, blank character, has different properties from, for example, –t "<space>". If a line contains:

```
<space><space>foo
```

the following treatment would occur with default separation as opposed to specifically selecting a space character:

Field	Default	–t "<space>"
1	<space><space>foo	*empty*
2	*empty*	*empty*
3	*empty*	foo

The leading field separator itself is included in a field when **–t** is not used. For example, this command returns an exit status of zero, meaning the input was already sorted:

```
sort -c -k 2 <<eof
y<tab>b
x<space>a
eof
```

(assuming that a tab character precedes the space character in the current collating sequence). The field separator is not included in a field when it is explicitly set via **–t**. This is historical practice and allows usage such as:

```
sort -t "|" -k 2n <<eof
Atlanta|425022|Georgia
Birmingham|284413|Alabama
Columbia|100385|South Carolina
eof
```

where the second field can be correctly sorted numerically without regard to the non-numeric field separator.

The wording in **OPTIONS** clarifies that the –**b**, –**d**, –**f**, –**i**, –**n** and –**r** options have to come before the first sort key specified if they are intended to apply to all specified keys. The way it is described in this document matches historical practice, not historical documentation. In the non-obsolescent versions, the results are unspecified if these options are specified after a –**k** option.

The –**f** option might not work as expected in locales where there is not a one-to-one mapping between an upper- and a lower-case letter.

EXAMPLES

In the following examples, non-obsolescent and obsolescent ways of specifying sort keys are given as an aid to understanding the relationship between the two forms.

1. Either of the following commands sorts the contents of **infile** with the second field as the sort key:

```
sort -k 2,2 infile
OB sort +1 -2 infile
```

2. Either of the following commands sorts, in reverse order, the contents of **infile1** and **infile2**, placing the output in **outfile** and using the second character of the second field as the sort key (assuming that the first character of the second field is the field separator):

```
sort -r -o outfile -k 2.2,2.2 infile1 infile2
OB sort -r -o outfile +1.1 -1.2 infile1 infile2
```

3. Either of the following commands sorts the contents of **infile1** and **infile2** using the second non-blank character of the second field as the sort key:

```
sort -k 2.2b,2.2b infile1 infile2
OB sort +1.1b -1.2b  infile1 infile2
```

4. Either of the following commands prints the System V password file (user database) sorted by the numeric user ID (the third colon-separated field):

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

5. Either of the following commands prints the lines of the already sorted file **infile**, suppressing all but one occurrence of lines having the same third field:

```
sort -um -k 3.1,3.0 infile
OB sort -um +2.0 -3.0 infile
```

FUTURE DIRECTIONS

None.

SEE ALSO

comm, *join*, *uniq*, the **XSH** specification description of *toupper*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

spell – find spelling errors (**TO BE WITHDRAWN**)

SYNOPSIS

EX `spell [-bvx][+local_file][file...]`

DESCRIPTION

The *spell* utility collects words from the named files and looks them up in a spelling list. A *word* in this context is a series of characters from the set:

```
[A-Za-z0-9'&.,;?:]
```

in the POSIX locale, where the first and last characters are alphanumeric. Words that neither occur among nor are derivable (by applying certain inflections, prefixes and suffixes) from words in the spelling list are written to standard output.

Within the file, certain character sequences are treated specially; if the file contains lines that begin with a period or apostrophe or that contain backslashes in any position, the results are unspecified.

OPTIONS

The *spell* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the *+local_file* option takes a leading plus sign instead of minus. The following options are supported:

–v Write all words not literally in the spelling list. Plausible derivations from the words in the spelling list may be indicated.

–b Check British spelling. Besides preferring *centre*, *colour*, *programme*, *speciality*, *travelled*, and so forth, this option insists upon *-ise* in words like *standardise*.

–x Write every plausible stem with = for each word.

+local_file

Remove words found in *local_ file* from the *spell* command 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.

OPERANDS

The following operands are supported:

file A pathname of a text file to check for spelling errors. If no files are named, words are collected from the standard input.

STDIN

The standard input is a text file used only if no *file* operands are specified.

INPUT FILES

The input files are text files.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *spell*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output consists of single misspelled words separated by newlines. If the **–x** option is used, words can be preceded by =.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None of the internationalisation variables are required to affect the processing of the input source language. In the POSIX locale, the *spell* utility recognises English (American or British dialects, depending on the **–b** option) text.

The unspecified nature of *spell* when presented files with backslashes or leading periods/apostrophes results from its complex attempts to deal with files formatted for *troff*, *tbl* and *eqn* processing (none of which are specified by this document). Constructs such as .so, .nx, .TS, .EQ, .ig, \s and \f are frequently dealt with in a manner that is most useful for determining spelling errors. However, such algorithms are historically less than perfect and are very difficult to describe precisely.

EXAMPLES
None.

FUTURE DIRECTIONS
The *spell* utility is being withdrawn because there is no known technology that can be used to make it recognise general language for user-specified input without providing a complete dictionary along with the input file.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

split – split files into pieces

SYNOPSIS

```
split [-l line_count][-a suffix_length][file[name]]
split -b n[k|m][-a suffix_length][file[name]]
OB split [-line_count][-a suffix_length][file[name]]
```

DESCRIPTION

The *split* utility reads an input file and writes one or more output files. The default size of each output file is 1000 lines. The size of the output files can be modified by specification of the **–b** or **–l** options. Each output file is created with a unique suffix. The suffix consists of exactly *suffix_length* lower-case letters from the POSIX locale. The letters of the suffix are used as if they were a base-26 digit system, with the first suffix to be created consisting of all a characters, the second with a b replacing the last a, and so forth, until a name of all z characters is created. By default, the names of the output files are x, followed by a two-character suffix from the character set as described above, starting with aa, ab, ac, and so forth, and continuing until the suffix zz, for a maximum of 676 files.

If the number of files required exceeds the maximum allowed by the suffix length provided, such that the last allowable file would be larger than the requested size, the *split* utility will fail after creating the last file with a valid suffix; *split* will not delete the files it created with valid suffixes. If the file limit is not exceeded, the last file created will contain the remainder of the input file, and may be smaller than the requested size.

OPTIONS

OB The *split* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** except that the obsolescent version allows a multi-digit option, *–line_count*.

The following options are supported:

–a *suffix_length*
: Use *suffix_length* letters to form the suffix portion of the filenames of the split file. If **–a** is not specified, the default suffix length is two. If the sum of the *name* operand and the *suffix_length* option-argument would create a filename exceeding {NAME_MAX} bytes, an error will result; *split* will exit with a diagnostic message and no files will be created.

–b *n*
: Split a file into pieces *n* bytes in size.

–b *n***k**
: Split a file into pieces *n**1024 bytes in size.

–b *n***m**
: Split a file into pieces *n**1 048 576 bytes in size.

–l *line_count*

OB *–line_count*
: Specify the number of lines in each resulting file piece. The *line_count* argument is an unsigned decimal integer. The default is 1000. If the input does not end with a newline character, the partial line will be included in the last output file.

OPERANDS

The following operands are supported:

file
: The pathname of the ordinary file to be split. If no input file is given or *file* is –, the standard input will be used.

name The prefix to be used for each of the files resulting from the split operation. If no *name* argument is given, x will be used as the prefix of the output files. The combined length of the basename of *prefix* and *suffix_length* cannot exceed {NAME_MAX} bytes; see **OPTIONS**.

STDIN

See **INPUT FILES**.

INPUT FILES

Any file can be used as input.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *split*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files contain portions of the original input file, otherwise unchanged.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

In the following examples foo is a text file that contains 5000 lines.

1. Create five files, xaa, xab, xac, xad and xae:

```
split foo
```

2. Create five files, but the suffixed portion of the created files consists of three letters, xaaa, xaab, xaac, xaad and xaae:

```
split -a 3 foo
```

3. Create three files with four-letter suffixes and a supplied prefix, bar_aaaa, bar_aaab and bar_aaac:

```
split -a 4 -l 2000 foo bar_
```

4. Create as many files as are necessary to contain at most 20*1024 bytes, each with the default prefix of **x** and a five-letter suffix:

```
split -a 5 -b 20k foo
```

FUTURE DIRECTIONS

None.

SEE ALSO

csplit.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

strings – find printable strings in files

SYNOPSIS

```
strings [-a][-t format][-n number][ile...]
```

OB

```
strings [-][-t format][-number][file...]
```

DESCRIPTION

The *strings* utility looks for printable strings in regular files and writes those strings to standard output. A printable string is any sequence of four (by default) or more printable characters terminated by a newline or NUL character. Additional implementation-dependent strings may be written. (See *localedef*.)

OPTIONS

OB The *strings* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** except that the obsolescent version uses – in a non-standard way and allows a multi-digit option, –*number*.

The following options are supported:

–**a**

OB – Scan files in their entirety. If –**a** is not specified, it is implementation-dependent what portion of each file is scanned for strings.

–**n** *number*

OB –*number*

Specify the minimum string length, where the *number* argument is a positive decimal integer. The default is 4.

–**t** *format*

Write each string preceded by its byte offset from the start of the file. The format is dependent on the single character used as the *format* option-argument:

d The offset will be written in decimal.

o The offset will be written in octal.

x The offset will be written in hexadecimal.

OPERANDS

The following operand is supported:

file A pathname of a regular file to be used as input. If no *file* operand is specified, the *strings* utility will read from the standard input.

STDIN

See **INPUT FILES**.

INPUT FILES

The input files named by the utility arguments or the standard input must be regular files of any format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *strings*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and to identify printable strings.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Strings found are written to the standard output, one per line.

When the **–t** option is not specified, the format of the output is:

```
"%s", <string>
```

With the **–t o** option, the format of the output is:

```
"%o %s", <byte offset>, <string>
```

With the **–t x** option, the format of the output is:

```
"%x %s", <byte offset>, <string>
```

With the **–t d** option, the format of the output is:

```
"%d %s", <byte offset>, <string>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
By default the data area (as opposed to the text, ‘‘bss’’ or header areas) of a binary executable file is scanned. Implementations will document which areas are scanned.

Some historical implementations do not require NUL or newline character terminators for strings to permit those languages that do not use NUL as a string terminator to have their strings written.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

nm.

CHANGE HISTORY

First released in Issue 4.

NAME

strip – remove unnecessary information from executable files (**DEVELOPMENT**)

SYNOPSIS

```
strip file...
```

DESCRIPTION

The *strip* utility removes from executable files named by the *file* operands any information the implementor deems unnecessary for execution of those files. The nature of that information is unspecified. The effect of *strip* is the same as the use of the **–s** option to *cc*, *c89* or *fort77*.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname referring to an executable file.

STDIN

Not used.

INPUT FILES

The input files must be in the form of executable files successfully produced by any compiler defined by this document.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *strip*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The *strip* utility will produce executable files of unspecified format.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ar, *cc*, *c89*, *fort77*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

stty – set the options for a terminal

SYNOPSIS

```
stty [ -a| -g]
stty operands
```

DESCRIPTION

The *stty* utility sets or reports on terminal I/O characteristics for the device that is its standard input. Without options or operands specified, it reports the settings of certain characteristics, usually those that differ from implementation-dependent defaults. Otherwise, it modifies the terminal state according to the specified operands. Detailed information about the modes listed in the first five groups below are described in the **XBD** specification, **Chapter 9**, **General Terminal Interface**. Operands in the Combination Modes group (see **Combination Modes** on page 670) are implemented using operands in the previous groups. Some combinations of operands are mutually exclusive on some terminal types; the results of using such combinations are unspecified.

Typical implementations of this utility require a communications line configured to use a **XSH** specification **termios** interface. On systems where none of these lines are available, and on lines not currently configured to support the **XSH** specification termios interface, some of the operands need not affect terminal characteristics.

OPTIONS

The *stty* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Write to standard output all the current settings for the terminal.

–g Write to standard output all the current settings in an unspecified form that can be used as arguments to another invocation of the *stty* utility on the same system. The form used will not contain any characters that would require quoting to avoid word expansion by the shell; see Section 2.6 on page 31.

OPERANDS

The following operands are supported to set the terminal characteristics:

Control Modes

parenb (–parenb)
Enable (disable) parity generation and detection. This has the effect of setting (not setting) PARENB in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

parodd (–parodd)
Select odd (even) parity. This has the effect of setting (not setting) PARODD in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

cs5 cs6 cs7 cs8
Select character size, if possible. This has the effect of setting CS5, CS6, CS7 and CS8, respectively, in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

number Set terminal baud rate to the number given, if possible. If the baud rate is set to zero, the modem control lines will no longer be asserted. This has the effect of setting the input and output **termios** baud rate values as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ispeed *number*
Set terminal input baud rate to the number given, if possible. If the input baud rate is set to zero, the input baud rate will be specified by the value of the output baud rate. This has the effect of setting the input **termios** baud rate values as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ospeed *number*
Set terminal output baud rate to the number given, if possible. If the output baud rate is set to zero, the modem control lines will no longer be asserted. This has the effect of setting the output **termios** baud rate values as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

hupcl (–hupcl)
Stop asserting modem control lines (do not stop asserting modem control lines) on last close. This has the effect of setting (not setting) HUPCL in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

hup (–hup)
Same as **hupcl** (–**hupcl**).

cstopb (–cstopb)
Use two (one) stop bits per character. This has the effect of setting (not setting) CSTOPB in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

cread (–cread)
Enable (disable) the receiver. This has the effect of setting (not setting) CREAD in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

clocal (–clocal)
Assume a line without (with) modem control. This has the effect of setting (not setting) CLOCAL in the **termios c_cflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

It is unspecified whether *stty* will report an error if an attempt to set a Control Mode fails.

Input Modes

ignbrk (–ignbrk)
Ignore (do not ignore) break on input. This has the effect of setting (not setting) IGNBRK in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

brkint (–brkint)
Signal (do not signal) INTR on break. This has the effect of setting (not setting) BRKINT in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ignpar (–ignpar)
Ignore (do not ignore) bytes with parity errors. This has the effect of setting (not setting) IGNPAR in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

parmrk (–parmrk)
Mark (do not mark) parity errors. This has the effect of setting (not setting) PARMRK in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

inpck (–inpck)
Enable (disable) input parity checking. This has the effect of setting (not setting) INPCK in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

istrip (–istrip)
Strip (do not strip) input characters to seven bits. This has the effect of setting (not setting) ISTRIP in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

inlcr (–inlcr)
Map (do not map) NL to CR on input. This has the effect of setting (not setting) INLCR in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

igncr (–igncr)
Ignore (do not ignore) CR on input. This has the effect of setting (not setting) IGNCR in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

icrnl (–icrnl)
Map (do not map) CR to NL on input. This has the effect of setting (not setting) ICRNL in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

iuclc (–iuclc)
EX Map (do not map) upper-case alphabetics to lower-case on input. This has the effect of setting (not setting) IUCLC in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**. **(TO BE WITHDRAWN)**

ixon (–ixon)
Enable (disable) START/STOP output control. Output from the system is stopped when the system receives STOP and started when the system receives START. This has the effect of setting (not setting) IXON in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ixany (–ixany)
EX Allow any character to restart output. This has the effect of setting (not setting) IXANY in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ixoff (–ixoff)
Request that the system send (not send) STOP characters when the input queue is nearly full and START characters to resume data transmission. This has the effect of setting (not setting) IXOFF in the **termios c_iflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

Output Modes

opost (–opost)
Post-process output (do not post-process output; ignore all other output modes). This has the effect of setting (not setting) OPOST in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

olcuc (–olcuc)
EX Map (do not map) lower-case alphabetics to upper-case on output. This has the effect of setting (not setting) OLCUC in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**. **(TO BE WITHDRAWN)**

ocrnl (–ocrnl)
EX Map (do not map) CR to NL on output This has the effect of setting (not setting) OCRNL in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

onocr (–onocr)
EX Do not (do) output CR at column zero. This has the effect of setting (not setting) ONOCR in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

onlret (–onlret)
EX The terminal newline key performs (does not perform) the CR function. This has the effect of setting (not setting) ONLRET in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ofill (–ofill)
EX Use fill characters (use timing) for delays. This has the effect of setting (not setting) OFILL in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

ofdel (–ofdel)
EX Fill characters are DELs (NULs). This has the effect of setting (not setting) OFDEL in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

cr0 cr1 cr2 cr3
EX Select the style of delay for CRs. This has the effect of setting (not setting) CRDLY to CR1, CR2, CR3 or CR4, respectively, in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

EX **nl0 nl1** Select the style of delay for NL. This has the effect of setting (not setting) NLDLY to NL0 or NL1, respectively, in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

tab0 tab1 tab2 tab3
EX Select the style of delay for horizontal tabs. This has the effect of setting (not setting) TABDLY to TAB0, TAB1, TAB2 or TAB3, respectively, in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**. Note that TAB3 has the effect of expanding tabs to spaces.

EX **bs0 bs1** Select the style of delay for backspaces. This has the effect of setting (not setting) BSDLY to BS0 or BS1, respectively, in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

EX **ff0 ff1** Select the style of delay for form-feeds. This has the effect of setting (not setting) FFDLY to FF0 or FF1, respectively, in the **termios** *c_oflag* field, as defined in the **XBD**

specification, **Chapter 9**, **General Terminal Interface**.

EX **vt0 vt1** Select the style of delay for vertical-tabs. This has the effect of setting (not setting) VTDLY to VT0 or VT1, respectively, in the **termios** *c_oflag* field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

Local Modes

isig (–isig)
Enable (disable) the checking of characters against the special control characters INTR, QUIT, and SUSP. This has the effect of setting (not setting) ISIG in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

icanon (–icanon)
Enable (disable) canonical input (ERASE and KILL processing). This has the effect of setting (not setting) ICANON in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

xcase (–xcase)
EX Set canonical (unprocessed) upper- or lower-case presentation. This has the effect of setting (not setting) XCASE in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**. **(TO BE WITHDRAWN)**

iexten (–iexten)
Enable (disable) any implementation-dependent special control characters not currently controlled by **icanon**, **isig**, **ixon** or **ixoff**. This has the effect of setting (not setting) IEXTEN in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

echo (–echo)
Echo back (do not echo back) every character typed. This has the effect of setting (not setting) ECHO in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

echoe (–echoe)
The ERASE character will (will not) visually erase the last character in the current line from the display, if possible. This has the effect of setting (not setting) ECHOE in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

echok (–echok)
Echo (do not echo) NL after KILL character. This has the effect of setting (not setting) ECHOK in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

echonl (–echonl)
Echo (do not echo) NL, even if **echo** is disabled. This has the effect of setting (not setting) ECHONL in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

noflsh (–noflsh)
Disable (enable) flush after INTR, QUIT, SUSP. This has the effect of setting (not setting) NOFLSH in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

tostop (–tostop)
Send SIGTTOU for background output. This has the effect of setting (not setting) TOSTOP in the **termios c_lflag** field, as defined in the **XBD** specification, **Chapter 9**,

General Terminal Interface.

Note: Setting TOSTOP has no effect on systems not supporting the job control option, but all XSI-conformant systems do support this option.

Special Control Character Assignments

<control>-character string

Set *<control>-character* to *string*. If *<control>-character* is one of the character sequences in the first column of the following table, the corresponding **XBD** specification, **Chapter 9**, **General Terminal Interface** control character from the second column will be recognised. This has the effect of setting the corresponding element of the **termios c_cc** array (see the **XSH** specification **<termios.h>**).

Control Character	c_cc Subscript	Description
eof	VEOF	EOF character
eol	VEOL	EOL character
erase	VERASE	ERASE character
intr	VINTR	INTR character
kill	VKILL	KILL character
quit	VQUIT	QUIT character
susp	VSUSP	SUSP character
start	VSTART	START character
stop	VSTOP	STOP character

Table 3-15 Control Character Names in *stty*

If *string* is a single character, the control character will be set to that character. If *string* is the two-character sequence ^– or the string undef, the control character will be set to {_POSIX_VDISABLE}, if it is in effect for the device; if {_POSIX_VDISABLE} is not in effect for the device, it will be treated as an error. In the POSIX locale, if *string* is a two-character sequence beginning with circumflex (^), and the second character is one of those listed in the ^*c* column of the following table, the control character will be set to the corresponding character value in the Value column of the table.

^*c*	Value	^*c*	Value	^*c*	Value
a, **A**	<SOH>	**l**, **L**	<FF>	**w**, **W**	<ETB>
b, **B**	<STX>	**m**, **M**	<CR>	**x**, **X**	<CAN>
c, **C**	<ETX>	**n**, **N**	<SO>	**y**, **Y**	<EM>
d, **D**	<EOT>	**o**, **O**	<SI>	**z**, **Z**	<SUB>
e, **E**	<ENQ>	**p**, **P**	<DLE>	**[**	<ESC>
f, **F**	<ACK>	**q**, **Q**	<DC1>	****	<FS>
g, **G**	<BEL>	**r**, **R**	<DC2>	**]**	<GS>
h, **H**	<BS>	**s**, **S**	<DC3>	**^**	<RS>
i, **I**	<HT>	**t**, **T**	<DC4>	**_**	<US>
j, **J**	<LF>	**u**, **U**	<NAK>	**?**	<DEL>
k, **K**	<VT>	**v**, **V**	<SYN>		

Table 3-16 Circumflex Control Characters in *stty*

min *number*
time *number*
Set the value of **min** or **time** to *number*. MIN and TIME are used in non-canonical mode input processing (–**icanon**).

Combination Modes

saved settings
Set the current terminal characteristics to the saved settings produced by the –**g** option.

evenp or **parity**
Enable **parenb** and **cs7**; disable **parodd**.

oddp Enable **parenb**, **cs7** and **parodd**.

–**parity**, –**evenp** or –**oddp**
Disable **parenb**, and set **cs8**.

EX **raw** (–**raw** or **cooked**)
Enable (disable) raw input and output. Raw mode is equivalent to setting:

```
stty cs8 erase ^- kill ^- intr ^- \
  quit ^- eof ^- eol ^- -opost -inpck
```

nl (–nl) Enable (disable) **icrnl**. In addition, –**nl** unsets **inlcr** and **igncr**.

EX **lcase (–lcase)**
Set (unset) **xcase**, **iuclc** and **olcuc** (**TO BE WITHDRAWN**).

EX **LCASE (–LCASE)**
Equivalent to lcase (–lcase) (**TO BE WITHDRAWN**).

EX **tabs** (–**tabs** or **tab8**)
Preserve tabs (expand to spaces) when printing. Equivalent to tab3.

ek Reset ERASE and KILL characters back to system defaults.

sane Reset all modes to some reasonable, unspecified, values.

STDIN

Although no input is read from standard input, standard input is used to get the current terminal I/O characteristics and to set new terminal I/O characteristics.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *stty*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
This variable will determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments)

and which characters are in the class **print**.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If operands are specified, no output is produced.

If the –**g** option is specified, *stty* will write to standard output the current settings in a form that can be used as arguments to another instance of *stty* on the same system.

If the –**a** option is specified, all of the information as described in **OPERANDS** will be written to standard output. Unless otherwise specified, this information is written as space-separated tokens in an unspecified format, on one or more lines, with an unspecified number of tokens per line. Additional information may be written.

If no options or operands are specified, an unspecified subset of the information written for the –**a** option is written.

If speed information is written as part of the default output, or if the –**a** option is specified and if the terminal input speed and output speed are the same, the speed information will be written as follows:

```
"speed %d baud;", <speed>
```

Otherwise, speeds will be written as:

```
"ispeed %d baud; ospeed %d baud;", <ispeed>, <ospeed>
```

In locales other than the POSIX locale, the word **baud** may be changed to something more appropriate in those locales.

If control characters are written as part of the default output, or if the –**a** option is specified, control characters will be written as:

```
"%s = %s;", <<control>-character name>, <value>
```

where *value* is either the character, or some visual representation of the character if it is non-printable, or the string <undef> if the character is disabled.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The terminal options were read or set successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The –**g** flag is designed to facilitate the saving and restoring of terminal state from the shell level. For example, a program may:

```
saveterm="$(stty -g)"   # save terminal state
stty (new settings)     # set new state
stty $saveterm          # restore terminal state
```

Since the format is unspecified, the saved value is not portable across systems.

Since the –**a** format is so loosely specified, scripts that save and restore terminal settings should use the –**g** option.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The **XBD** specification, **Chapter 9**, **General Terminal Interface**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

sum – print checksum and block count of a file (**TO BE WITHDRAWN**)

SYNOPSIS

EX `sum [-r][file...]`

DESCRIPTION

The *sum* utility calculates and writes a checksum for the named file to standard output and also writes the space used by the file, in 512-byte units.

OPTIONS

The *sum* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

UN **–r** Use an alternative algorithm in computing the checksum.

OPERANDS

The following operands are supported:

file A pathname of a file. If no files are named, the standard input is read.

STDIN

The standard input is any type of file, used only if no *file* operands are specified.

INPUT FILES

The input files are of any file type.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *sum*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

For each file processed successfully, *sum* writes a line of the following format:

```
"%u %d %s\n", <checksum>, <number of 512-byte units>, <pathname>
```

If the standard input is used for input, the pathname is omitted.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

It is not clear that the algorithms used in typical implementations are portable, that is, the same checksum might not be produced for the same input on different systems. Portable applications should use *cksum*.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

cksum.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support made optional.

Marked **TO BE WITHDRAWN**.

NAME

tabs – set terminal tabs

SYNOPSIS

EX UN `tabs [ -`*n*`| -a| -a2| -c| -c2| -c3| -f| -p| -s| -u][+m[`*n*`]]`
`[-T` *type*`]`

EX `tabs [-T` *type*`][ +[`*n*`]]` *n1*`[,`*n2*`,...]`

DESCRIPTION

The *tabs* utility displays a series of characters that first clears the hardware terminal tab settings
EX and then initialises the tab stops at the specified positions and optionally adjusts the margin.

The phrase ‘‘tab-stop position *N*’’ is taken to mean that, from the start of a line of output, tabbing to position *N* will cause the next character output to be in the (*N*+1)th column position on that line. The maximum number of tab stops allowed is terminal-dependent.

It need not be possible to implement *tabs* on certain terminals. If the terminal type obtained from the *TERM* environment variable or **–T** option represents such a terminal, an appropriate diagnostic message will be written to standard error and *tabs* will exit with a status greater than zero.

OPTIONS

EX The *tabs* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except for various extensions: the options **–a2**, **–c2** and **–c3** are multi-character and +m**[***n***]** uses a leading plus sign and an optional option-argument.

The following options are supported:

–n Specify repetitive tab stops separated by a uniform number of column positions, *n*, where *n* is a single-digit decimal number. The default usage of *tabs* with no arguments is equivalent to tabs –8. When **–0** is used, the tab stops are cleared and no new ones set.

EX **–a** 1,10,16,36,72
Assembler, applicable to some mainframes.

EX **–a2** 1,10,16,40,72
Assembler, applicable to some mainframes.

EX **–c** 1,8,12,16,20,55
COBOL, normal format.

EX **–c2** 1,6,10,14,49
COBOL, compact format (columns 1–6 omitted).

EX **–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**.

EX **–f** 1,7,11,15,19,23
FORTRAN

EX **–p** 1,5,9,13,17,21,25,29,33,37,41,45,49,53,57,61
PL/1

EX **–s** 1,10,55
SNOBOL

EX **–u** 1,12,20,44
Assembler, applicable to some mainframes.

–T *type*
Indicate the type of terminal. If this option is not supplied and the *TERM* variable is unset or null, an unspecified default terminal type will be used. The setting of *type* will take precedence over the value in *TERM*.

EX UN **+m[***n***]**
Reset the margin. The margin argument can be used for some terminals. It causes all tabs to be moved over *n* columns by making column *n*+1 the left margin. If *n* is omitted, the default is 10. 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.

OPERANDS

The following operand is supported:

*n1***[,***n2*,...**]**
A single command-line argument that consists of tab-stop values separated using either commas or blank characters. The tab-stop values will be positive decimal integers in strictly ascending order. 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. For example, the tab lists 1,10,20,30 and 1,10,+10,+10 are considered to be identical.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tabs*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TERM Determine the terminal type. If this variable is unset or null, and if the **–T** option is not specified, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS

Default.

STDOUT

If standard output is a terminal, the appropriate sequence to clear and set the tab stops may be written to standard output in an unspecified format. If standard output is not a terminal, undefined results occur.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility makes use of the terminal's hardware tabs and the *stty tabs* option.

This utility is not recommended for application use.

Some integrated display units might not have escape sequences to set tab stops, but may be set by internal system calls. On these terminals, *tabs* will work if standard output is directed to the terminal; if output is directed to another file, however, *tabs* will fail.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

expand, *stty*, *unexpand*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

tail – copy the last part of a file

SYNOPSIS

```
tail [-f][ -c number| -n number] [file]
```

EX OB
```
tail -[number][b|c|l][f] [file]
```

EX OB
```
tail +[number][b|c|l][f] [file]
```

DESCRIPTION

The *tail* utility copies its input file to the standard output beginning at a designated place.

OB Copying begins at the point in the file indicated by the **–c** *number* or **–n** *number* options (or the ±*number* portion of the argument to the obsolescent version). The option-argument *number* is
EX OB counted in units of lines or bytes, according to the options **–n** and **–c** (or, in the obsolescent version, the appended option suffixes l (lines), b (512-byte blocks) or c (bytes)). Both line and byte counts start from 1.

Tails relative to the end of the file may be saved in an internal buffer, and thus may be limited in length. Such a buffer, if any, will be no smaller than {LINE_MAX}*10 bytes.

OPTIONS

OB The *tail* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the obsolescent version accepts multi-character options that can preceded by a plus sign.

The following options are supported:

–c *number*

The *number* option-argument must be a decimal integer whose sign affects the location in the file, measured in bytes, to begin the copying:

Sign	Copying Starts
+	Relative to the beginning of the file.
–	Relative to the end of the file.
none	Relative to the end of the file.

The origin for counting is 1; that is, **–c +1** represents the first byte of the file, **–c –1** the last.

–f If the input file is a regular file or if the *file* operand specifies a FIFO, do not terminate after the last line of the input file has been copied, but read and copy further bytes from the input file when they become available. If no *file* operand is specified and standard input is a pipe, the **–f** option will be ignored. If the input file is not a FIFO, pipe or regular file, it is unspecified whether or not the **–f** option will be ignored.

–n *number*

This option is equivalent to **–c** *number*, except the starting location in the file is measured in lines instead of bytes. The origin for counting is 1; that is, **–n +1** represents the first line of the file, **–n –1** the last.

In the non-obsolescent form, if neither **–c** nor **–n** is specified, **–n 10** is assumed.

OB In the obsolescent version, an argument beginning with a – or + can be used as a single option. The argument ±*number* with the letter c specified as a suffix is equivalent to **–c** ±*number*; ±*number* with the b suffix is equivalent to **–c** ±*number**512; ±*number* with the letter l specified as a suffix, or with none of b, c nor l as a suffix, is equivalent to **–n** ±*number*. If *number* is not specified in these forms, 10 will be used. The letter f specified as a suffix is equivalent to specifying the **–f** option. If the **–[***number***]c[f]** form is used and neither *number* nor the f suffix is specified, it will be interpreted as the **–c 10** option.

OPERANDS

The following operand is supported:

file A pathname of an input file. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

If the **–c** option is specified, the input file can contain arbitrary data; otherwise, the input file must be a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tail*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The designated portion of the input file will be written to standard output.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The **–c** option should be used with caution when the input is a text file containing multi-byte characters; it may produce output that does not start on a character boundary.

Although the input file to *tail* can be any type, the results might not be what would be expected on some character special device files or on file types not described by the **XSH** specification. Since this document does not specify the block size used when doing input, *tail* need not read all of the data from devices that only perform block transfers.

EX OB The b suffix in the obsolescent version is not portable outside of X/Open systems.

EXAMPLES

The **–f** option can be used to monitor the growth of a file that is being written by some other process. For example, the command:

```
tail –f fred
```

prints 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 –f -c 15 fred
```

prints the last 15 bytes of the file **fred**, followed by any bytes that are appended to **fred** between the time **tail** is initiated and killed.

FUTURE DIRECTIONS

None.

SEE ALSO

head.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO/IEC 9945-2: 1993 standard.

NAME

talk – talk to another user

SYNOPSIS

`talk` *address* **[***terminal***]**

DESCRIPTION

The *talk* utility is a two-way, screen-oriented communication program.

When first invoked, *talk* sends a message similar to:

```
Message from <unspecified string>
talk: connection requested by your_address
talk: respond with: talk your_address
```

to the specified *address*. At this point, the recipient of the message can reply by typing:

```
talk  your_address
```

Once communication is established, the two parties can type simultaneously, with their output displayed in separate regions of the screen. Characters are processed as follows:

- Typing the alert character will alert the recipient's terminal.
- Typing <control>-L will cause the sender's screen regions to be refreshed.
- Typing the erase and kill characters will affect the sender's terminal in the manner described by the **termios** interface in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.
- Typing the interrupt or end-of-file characters will terminate the local *talk* utility. Once the *talk* session has been terminated on one side, the other side of the *talk* session will be notified that the *talk* session has been terminated and will be able to do nothing except exit.
- Typing characters from LC_CTYPE classifications **print** or **space** will cause those characters to be sent to the recipient's terminal.
- When and only when the *stty iexten* local mode is enabled, the existence and processing of additional special control characters and multi-byte or single-byte functions is implementation-dependent.
- Typing other non-printable characters will cause implementation-dependent sequences of printable characters to be sent to the recipient's terminal.

Permission to be a recipient of a *talk* message can be denied or granted by use of the *mesg* utility. However, a user's privilege may further constrain the domain of accessibility of other users' terminals. The *talk* utility will fail when the user lacks the appropriate privileges to perform the requested action.

Certain block-mode terminals do not have all the capabilities necessary to support the simultaneous exchange of messages required for *talk*. When this type of exchange cannot be supported on such terminals, the implementation may support an exchange with reduced levels of simultaneous interaction or it may report an error describing the terminal-related deficiency.

OPTIONS

None.

OPERANDS

The following operands are supported:

address The recipient of the *talk* session. One form of *address* is the <*user name*>, as returned by the *who* utility. Other address formats and how they are handled are unspecified.

terminal

If the recipient is logged in more than once, the *terminal* argument can be used to indicate the appropriate terminal name. If *terminal* is not specified, the *talk* message will be displayed on one or more accessible terminals in use by the recipient. The format of *terminal* will be the same as that returned by the *who* utility.

STDIN

Characters read from standard input will be copied to the recipient's terminal in an unspecified manner. If standard input is not a terminal, talk will write a diagnostic message and exit with a non-zero status.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *talk*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files). If the recipient's locale does not use an LC_CTYPE equivalent to the sender's, the results are undefined.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TERM Determine the name of the invoker's terminal type. If this variable is unset or null, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS

When the *talk* utility receives a SIGINT signal, the utility will terminate and exit with a zero status. It will take the standard action for all other signals.

STDOUT

If standard output is a terminal, characters copied from the recipient's standard input may be written to standard output. Standard output also may be used for diagnostic messages. If standard output is not a terminal, *talk* will exit with a non-zero status.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred or *talk* was invoked on a terminal incapable of supporting it.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Because the handling of non-printable, non-space characters is tied to the *stty* description of *iexten*, implementation extensions within the terminal driver can be accessed. For example, some implementations provide line editing functions with certain control character sequences.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg, *who*, *write*, the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

CHANGE HISTORY

First released in Issue 4.

NAME

tar – file archiver (**TO BE WITHDRAWN**)

SYNOPSIS

EX `tar key [file...]`

DESCRIPTION

The *tar* utility processes archives of files. Its actions are controlled by the *key* operand.

OPTIONS

None.

OPERANDS

The following operands are supported:

key The *key* operand consists of a function letter followed immediately by zero or more modifying letters.

The function letter is one of the following:

r Write the named *file* or files on the end of the archive. If the archive is on a magnetic tape device, the results are unspecified.

x Extract the named *file* or files 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 has appropriate privileges. 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 *file* operand 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 Write to standard output the names of all the files in the archive.

u Add the named *file* or files to the archive if they are not already there, or have been modified since last written into the archive. If the archive is on a magnetic tape device, the results are unspecified.

c Create a new archive; writing begins at the beginning of the archive, instead of after the last file.

The following characters can be appended to the function letter. Appending the same character more than once produces undefined results. The order of the b and f characters is significant.

v (Verbose.) Write to standard error the name of each file processed, preceded by a string indicating the operation being performed, as follows:

Key Letter	String
c, r, u	"a "
x	"x "

The filename may be followed by additional information, such as the size of the file in the archive or file system, in an unspecified format. When used with the t function letter, v writes to standard output more information about the archive entries than just the name.

w Write the action to be taken, followed by the name of the file, and then wait for the user's confirmation. If an affirmative response is given, the action is performed. Any other input suppresses the action.

f Use the first *file* operand (or the second, if b has already been specified) as the name of the archive instead of the system-dependent default. 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 *tar* utility can also be used to move directory hierarchies with the command:

```
(cd fromdir; tar cf – . ) | (cd todir; tar xf –)
```

b Use the first *file* operand (or the second, if f has already been specified) as the blocking factor for tape records. The default is not greater than 20; the maximum is not less than 20. This modifier should only be used with raw magnetic tape archives (see f above). The block size is determined automatically when reading tapes (function letters x and t).

l Report if all of the links to the files being archived cannot be resolved. If l is not specified, no error messages are written.

m Do not restore the modification times. The modification time of the file will be the time of extraction.

o Assign to extracted files the user and group identifier of the user running the program rather than those on the archive.

file A pathname of a regular file or directory to be archived (when the c, r or u function letters are used), extracted (x) or listed (t). When *file* is the pathname of a directory, the action applies to all of the files and (recursively) subdirectories of that directory. When either or both of the b or f letters are used in the *key* operand, the initial *file* operands are interpreted as a blocking factor or archive name, as described previously.

STDIN

When the f modifier is used with the t or x function letter and the pathname is –, the standard input is an archive file formatted as specified by *pax* with the **–x ustar** option. Otherwise, the standard input is not used.

INPUT FILES

The files identified by the *file* operands are regular files or directories.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *tar*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES
Determine the locale for the processing of affirmative responses and that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the format of date and time strings output when listing the contents of an archive with the v modifier; for example:

```
tar tvf /dev/tape
```

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When the f modifier is used with the r, u or c function letter and the pathname is –, the standard output is an archive file formatted as specified by *pax* with the **–x ustar** option. When the t function letter is used, the standard output consists of the names of the files in the archive, separated by newline characters; if v is used with t, the standard output includes additional information in an unspecified format. Otherwise, the standard output is not used.

STDERR
The standard error is used for diagnostic messages and the filename output described under the v modifier (when the t function letter is not used).

OUTPUT FILES
Output files are created, as specified by the archive, when the x function letter is used.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

EXAMPLES
None.

APPLICATION USAGE
Some systems have usually had blocking factors in the range 1 to at least 127 with a default of 20 while other systems have usually had blocking factors in the range 1 to 20 with a default of 1. For maximum portability, applications should specify a blocking factor no larger than 20.

For portable communication of data between X/Open systems, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the Portable Filename Character Set be used for naming files. This recommendation is given because X/Open systems support diverse codesets and run in various geographical areas and there is no single, well established codeset that incorporates all of the characters of the languages of the various geographical areas.

FUTURE DIRECTIONS

This utility is being withdrawn from a future issue. The *pax* utility should be used instead.

SEE ALSO

cpio, *pax*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The behaviour of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Marked **TO BE WITHDRAWN**.

Internationalised environment variable support made optional.

NAME

tee – duplicate standard input

SYNOPSIS

```
tee [-ai][file...]
```

DESCRIPTION

The *tee* utility will copy standard input to standard output, making a copy in zero or more files. The *tee* utility will not buffer output.

The options determine if the specified files are overwritten or appended to.

OPTIONS

The *tee* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Append the output to the files rather than overwriting them.

–i Ignore the SIGINT signal.

OPERANDS

The following operands are supported:

file A pathname of an output file. Processing of at least 13 *file* operands will be supported.

STDIN

The standard input can be of any type.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tee*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default, except that if the **–i** option was specified, SIGINT will be ignored.

STDOUT

The standard output will be a copy of the standard input.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

If any *file* operands are specified, the standard input will be copied to each named file.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The standard input was successfully copied to all output files.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If a write to any successfully opened *file* operand fails, writes to other successfully opened *file* operands and standard output will continue, but the exit status will be non-zero. Otherwise, the default actions specified in the **XBD** specification, **Section 2.1**, **Utility Description Defaults** will apply.

APPLICATION USAGE

The *tee* utility is usually used in a pipeline, to make a copy of the output of some utility.

The *file* operand is technically optional, but *tee* is no more useful than *cat* when none is specified.

EXAMPLES

Save an unsorted intermediate form of the data in a pipeline:

```
... | tee unsorted | sort > sorted
```

FUTURE DIRECTIONS

None.

SEE ALSO

cat.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

test – evaluate expression

SYNOPSIS

`test [`*`expression`*`]`

`[ [`*`expression`*`] ]`

DESCRIPTION

The *test* utility evaluates the *expression* and indicates the result of the evaluation by its exit status. An exit status of zero indicates that the expression evaluated as true and an exit status of 1 indicates that the expression evaluated as false.

In the second form of the utility, which uses [ ] rather than *test*, the square brackets must be separate arguments.

OPTIONS

The *test* utility does not recognise the – – argument in the manner specified by guideline 10 in the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

No options are supported.

OPERANDS

All operators and elements of primaries must be presented as separate arguments to the *test* utility.

The following primaries can be used to construct *expression*:

–**b** *file* True if *file* exists and is a block special file.

–**c** *file* True if *file* exists and is a character special file.

–**d** *file* True if *file* exists and is a directory.

–**e** *file* True if *file* exists.

–**f** *file* True if *file* exists and is a regular file.

–**g** *file* True if *file* exists and its set group ID flag is set.

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

–**p** *file* True if *file* is a named pipe (FIFO).

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

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

–**t** *file_descriptor*
True if the file whose file descriptor number is *file_descriptor* is open and is associated with a terminal.

–**u** *file* True if *file* exists and its set-user-ID flag is set.

–**w** *file* True if *file* exists and is writable. True will indicate only that the write flag is on. The *file* will not be writable on a read-only file system even if this test indicates true.

–**x** *file* True if *file* exists and is executable. True will indicate only that the execute flag is on. If *file* is a directory, true indicates that *file* can be searched.

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

string True if the string *string* is not the null string.

s1 **=** *s2* True if the strings *s1* and *s2* are identical.

s1 **!=** *s2* True if the strings *s1* and *s2* are not identical.

n1 **–eq** *n2*
True if the integers *n1* and *n2* are algebraically equal.

n1 **–ne** *n2*
True if the integers *n1* and *n2* are not algebraically equal.

n1 **–gt** *n2*
True if the integer *n1* is algebraically greater than the integer *n2*.

n1 **–ge** *n2*
True if the integer *n1* is algebraically greater than or equal to the integer *n2*.

n1 **–lt** *n2*
True if the integer *n1* is algebraically less than the integer *n2*.

n1 **–le** *n2*
True if the integer *n1* is algebraically less than or equal to the integer *n2*.

EX *expression1* **–a** *expression2*
True if both *expression1* and *expression2* are true. The **–a** binary primary is left associative. It has a higher precedence than **–o**.

EX *expression1* **–o** *expression2*
True if either *expression1* or *expression2* is true. The **–o** binary primary is left associative.

These primaries can be combined with the following operators:

! *expression*
True if *expression* is false.

EX **(** *expression* **)**
True if *expression* is true. The parentheses can be used to alter the normal precedence and associativity.

The primaries with two elements of the form:

–primary_operator primary_operand

are known as *unary primaries.* The primaries with three elements in either of the two forms:

primary_operand –primary_operator primary_operand

primary_operand primary_operator primary_operand

are known as *binary primaries.* Additional implementation-dependent operators and *primary_operator*s may be provided by implementations. They will be of the form *–operator* where the first character of *operator* is not a digit.

The algorithm for determining the precedence of the operators and the return value that will be generated is based on the number of arguments presented to *test*. (However, when using the [...] form, the right-bracket final argument will not be counted in this algorithm.)

In the following list, $1, $2, $3 and $4 represent the arguments presented to *test*.

0 arguments: Exit false (1).

1 argument: Exit true (0) if $1 is not null; otherwise, exit false.

2 arguments:

- If $1 is !, exit true if $2 is null, false if $2 is not null.
- If $1 is a unary primary, exit true if the unary test is true, false if the unary test is false.
- Otherwise, produce unspecified results.

3 arguments:

- If $2 is a binary primary, perform the binary test of $1 and $3.
- If $1 is !, negate the two-argument test of $2 and $3.
- EX If $1 is (and $3 is), perform the unary test of $2.
- Otherwise, produce unspecified results.

4 arguments:

- If $1 is !, negate the three-argument test of $2, $3 and $4.
- EX If $1 is (and $4 is), perform the two-argument test of $2 and $3.
- Otherwise, the results are unspecified.

EX >4 arguments: Combinations of primaries and operators are evaluated using the precedence and associativity rules described previously. In addition, the string comparison binary primaries = and != have a higher precedence than any unary primary.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *test*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 *expression* evaluated to true.
1 *expression* evaluated to false or *expression* was missing.
>1 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Scripts should be careful when dealing with user-supplied input that could be confused with primaries and operators. Unless the application writer knows all the cases that produce input to the script, invocations like:

```
test "$1" -a "$2"
```

should be written as:

```
test "$1" && test "$2"
```

to avoid problems if a user supplied values such as $1 set to ! and $2 set to the null string. That is, in cases where maximal portability is of concern, replace:

```
test expr1 -a expr2
```

with:

```
test expr1 && test expr2
```

and replace:

```
test expr1 -o expr2
```

with:

```
test expr1 || test expr2
```

but note that, in *test*, –**a** has higher precedence than –**o** while && and | | have equal precedence in the shell.

Parentheses or braces can be used in the shell command language to effect grouping.

Parentheses must be escaped when using *sh*; for example:

```
test \( expr1 -a expr2 \) -o expr3
```

This command is not always portable outside XSI-conformant systems. The following form can be used instead:

```
( test expr1 && test expr2 ) || test expr3
```

The two commands:

```
test "$1"
test ! "$1"
```

could not be used reliably on some historical systems. Unexpected results would occur if such a *string* expression were used and $1 expanded to !, (or a known unary primary. Better constructs are:

```
test -n "$1"
test -z "$1"
```

respectively.

Historical systems have also been unreliable given the common construct:

```
test "$response" = "expected string"
```

One of the following is a more reliable form:

```
test "X$response" = "Xexpected string"
test "expected string" = "$response"
```

Note that the second form assumes that expected string could not be confused with any unary primary. If expected string starts with –, (, ! or even =, the first form should be used instead. Using the preceding rules without the marked extensions, any of the three comparison forms is reliable, given any input. (However, note that the strings are quoted in all cases.)

Because the string comparison binary primaries, = and !=, have a higher precedence than any unary primary in the >4 argument case, unexpected results can occur if arguments are not properly prepared. For example, in:

```
test -d $1 -o -d $2
```

If $1 evaluates to a possible directory name of **=**, the first three arguments are considered a string comparison, which causes a syntax error when the second **–d** is encountered. One of the following forms prevents this; the second is preferred:

```
test \( -d "$1" \) -o \( -d "$2" \)
test -d "$1" || test -d "$2"
```

Also in the >4 argument case:

```
test "$1" = "bat" -a "$2" = "ball"
```

Syntax errors will occur if $1 evaluates to (or !. One of the following forms prevents this; the third is preferred:

```
test "X$1" = "Xbat" -a "X$2" = "Xball"
test "$1" = "bat" && test "$2" = "ball"
test "X$1" = "Xbat" && test "X$2" = "Xball"
```

EXAMPLES

EX

1. Exit if there are not two or three arguments (two variations):

```
if [ $# -ne 2 -a $# -ne 3 ]; then exit 1; fi
if [ $# -lt 2 -o $# -gt 3 ]; then exit 1; fi
```

2. Perform a *mkdir* if a directory does not exist:

```
test ! -d tempdir && mkdir tempdir
```

3. Wait for a file to become non-readable:

```
while test -r thefile
do
       sleep 30
done
echo '"thefile" is no longer readable'
```

4. Perform a command if the argument is one of three strings (two variations):

```
if [ "$1" = "pear" ] || [ "$1" = "grape" ] || [ "$1" = "apple" ]
then
       command
fi

case "$1" in
       pear|grape|apple) command ;;
esac
```

FUTURE DIRECTIONS

None.

SEE ALSO

find.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

time – time a simple command

SYNOPSIS

`time [-p] utility [argument...]`

DESCRIPTION

The *time* utility invokes the utility named by the *utility* operand with arguments supplied as the *argument* operands and writes a message to standard error that lists timing statistics for the utility. The message includes the following information:

- The elapsed (real) time between invocation of *utility* and its termination.
- The User CPU time, equivalent to the sum of the *tms_utime* and *tms_cutime* fields returned by the **XSH** specification *times*( ) function for the process in which *utility* is executed.
- The System CPU time, equivalent to the sum of the *tms_stime* and *tms_cstime* fields returned by the *times*( ) function for the process in which *utility* is executed.

The precision of the timing will be no less than the granularity defined for the size of the clock tick unit on the system, but the results will be reported in terms of standard time units (for example, 0.02 seconds, 00:00:00.02, 1m33.75s, 365.21 seconds), not numbers of clock ticks.

When *time* is used as part of a pipeline, the times reported are unspecified, except when it is the sole command within a grouping command (see Section 2.9.4 on page 52) in that pipeline. For example, the commands on the left are unspecified; those on the right report on utilities a and c, respectively.

```
time a | b | c             { time a } | b | c
a | b | time c             a | b | (time c)
```

OPTIONS

The *time* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–p Write the timing output to standard error in the format shown in **STDERR**.

OPERANDS

The following operands are supported:

utility The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

argument
Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *time*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic and informative messages written to standard error.

LC_NUMERIC
Determine the locale for numeric formatting.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the search path that will be used to locate the utility to be invoked. See the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
The standard error will be used to write the timing statistics. If **–p** is specified, the following format will be used in the POSIX locale:

```
"real %f\nuser %f\nsys %f\n", <real seconds>, <user seconds>,
<system seconds>
```

where each floating-point number is expressed in seconds. The precision used may be less than the default six digits of **%f**, but will be sufficiently precise to accommodate the size of the clock tick on the system (for example, if there were 60 clock ticks per second, at least two digits follow the radix character). The number of digits following the radix character will be no less than one, even if this always results in a trailing zero. The implementation may append white space and additional information following the format shown here.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
If the *utility* utility is invoked, the exit status of *time* will be the exit status of *utility;* otherwise, the *time* utility will exit with one of the following values: The following exit values are returned:

1–125 An error occurred in the *time* utility.

126 The utility specified by *utility* was found but could not be invoked.

127 The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE

The *command*, *env*, *nice*, *nohup*, *time*, and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ''failure to find a utility'' from ''invoked utility exited with an error indication.'' The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ''normal error conditions'' and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

It is frequently desirable to apply *time* to pipelines or lists of commands. This can be done by placing pipelines and command lists in a single file; this file can then be invoked as a utility, and the *time* applies to everything in the file.

Alternatively, the following command can be used to apply *time* to a complex command:

```
time sh -c 'complex-command-line'
```

FUTURE DIRECTIONS

None.

SEE ALSO

sh times special built-in utility.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

touch – change file access and modification times

SYNOPSIS

```
touch [-acm][ -r ref_file| -t time] file...
```

OB

```
touch [-acm][date_time] file...
```

DESCRIPTION

The *touch* utility will change the modification times, access times or both of files. The modification time is equivalent to the value of the *st_mtime* member of the *stat* structure for a file, as described in the **XSH** specification; the access time is equivalent to the value of *st_atime*.

The time used can be specified by the –**t** *time* option-argument, the corresponding time fields of the file referenced by the –**r** *ref_file* option-argument, or the *date_time* operand, as specified in the following sections. If none of these are specified, *touch* will use the current time (the value returned by the equivalent of the **XSH** specification *time*( ) function).

For each *file* operand, *touch* will perform actions equivalent to the following functions defined in the **XSH** specification:

1. If *file* does not exist, a *creat*( ) function call is made with the *file* operand used as the *path* argument and the value of the bitwise inclusive OR of S_IRUSR, S_IWUSR, S_IRGRP, S_IWGRP, S_IROTH and S_IWOTH used as the *mode* argument.
2. The *utime*( ) function is called with the following arguments:
   a. The *file* operand is used as the *path* argument.
   b. The *utimbuf* structure members *actime* and *modtime* are determined as described under **OPTIONS**.

OPTIONS

The *touch* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–**a** Change the access time of *file*. Do not change the modification time unless –**m** is also specified.

–**c** Do not create a specified *file* if it does not exist. Do not write any diagnostic messages concerning this condition.

–**m** Change the modification time of *file*. Do not change the access time unless –**a** is also specified.

–**r** *ref_file*
Use the corresponding time of the file named by the pathname *ref_file* instead of the current time.

–**t** *time*
Use the specified *time* instead of the current time. The option-argument will be a decimal number of the form:

[[*CC***]***YY***]***MMDDhhmm***[.***SS***]**

where each two digits represents the following:

MM The month of the year [01–12].

DD The day of the month [01–31].

hh The hour of the day [00–23].

mm The minute of the hour [00–59].

CC The first two digits of the year (the century).

YY The second two digits of the year.

SS The second of the minute [00–61].

Both *CC* and *YY* are optional. If neither is given, the current year will be assumed. If *YY* is specified, but *CC* is not, *CC* will be derived as follows:

If *YY* is:	*CC* becomes:
69–99	19
00–68	20

The resulting time will be affected by the value of the *TZ* environment variable. If the resulting time value precedes the Epoch, *touch* will exit immediately with an error status. The range of valid times past the Epoch is implementation-dependent, but will extend to at least midnight 1 January 2000 UCT. Some systems will not be able to represent dates beyond the January 18, 2038, because they use **signed int** as a time holder.

The range for *SS* is (00–61) rather than (00–59) because of leap seconds. If *SS* is 60 or 61, and the resulting time, as affected by the *TZ* environment variable, does not refer to a leap second, the resulting time will be one or two seconds after a time where *SS* is 59. If *SS* is not given a value, it is assumed to be zero.

If neither the **–a** nor **–m** options were specified, *touch* will behave as if both the **–a** and **–m** options were specified.

OPERANDS

The following operands are supported:

file A pathname of a file whose times are to be modified.

OB *date_time*

Use the specified *date_time* instead of the current time. The operand is a decimal number of the form:

MMDDhhmm[*yy*]

where *MM*, *DD*, *hh*, and *mm* are as described for the *time* option-argument to the **–t** option and the optional *yy* is interpreted as follows:

If not specified, the current year will be used. If *yy* is in the range 69–99, the year 1969–1999, respectively, will be used. Otherwise, the results are unspecified.

If no **–r** option is specified, no **–t** option is specified, at least two operands are specified, and the first operand is an eight- or ten-digit decimal integer, the first operand will be assumed to be a *date_time* operand. Otherwise, the first operand will be assumed to be a *file* operand.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *touch*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

OB *TZ* Determine the timezone to be used for interpreting the *time* option-argument (or *date_time* operand; see above).

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The utility executed successfully and all requested changes were made.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The interpretation of time is taken to be **seconds since the Epoch** (see the **XBD** specification, **Chapter 2**, **Glossary**). It should be noted that implementations conforming to the **XSH** specification do not take leap seconds into account when computing seconds since the Epoch. When *SS*=60 is used, the resulting time always refers to 1 plus ‘‘seconds since the Epoch’’ for a time when *SS*=59.

Although the **–t** *time* option-argument and the obsolescent *date_time* operand specify values in 1969, the access time and modification time fields are defined in terms of seconds since the Epoch (midnight on 1 January 1970 UTC). Therefore, depending on the value of *TZ* when *touch*

is run, there will never be more than a few valid hours in 1969 and there need not be any valid times in 1969.

One ambiguous situation occurs if **–t** *time* is not specified, **–r** *ref_file* is not specified, and the first operand is an eight- or ten-digit decimal number. A portable script can avoid this problem by using:

```
touch -- file
```

or:

```
touch ./file
```

in this case.

EXAMPLES

None.

FUTURE DIRECTIONS

The obsolescent *date_time* operand may be withdrawn from a future issue. Applications should use the **–r** or **–t** options.

SEE ALSO

date, the **XSH** specification description of *creat*( ), *time*( ), **<sys/stat.h>**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

tput – change terminal characteristics

SYNOPSIS

`tput [-T type] operand...`

DESCRIPTION

The *tput* utility displays terminal-dependent information. The manner in which this information is retrieved is unspecified. The information displayed will clear the terminal screen, initialise the user's terminal or reset the user's terminal, depending on the operand given. The exact consequences of displaying this information are unspecified.

OPTIONS

The *tput* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–T *type* Indicate the type of terminal. If this option is not supplied and the *TERM* variable is unset or null, an unspecified default terminal type will be used. The setting of *type* will take precedence over the value in *TERM*.

OPERANDS

The following strings will be supported as operands by the implementation in the POSIX locale:

clear Display the clear-screen sequence.

init Display the sequence that will initialise the user's terminal in an implementation-dependent manner.

reset Display the sequence that will reset the user's terminal in an implementation-dependent manner.

If a terminal does not support any of the operations described by these operands, this will not be considered an error condition.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tput*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TERM Determine the terminal type. If this variable is unset or null, and if the **–T** option is not specified, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS
Default.

STDOUT
If standard output is a terminal device, it may be used for writing the appropriate sequence to clear the screen or reset or initialise the terminal. If standard output is not a terminal device, undefined results occur.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The requested string was written successfully.
1 Unspecified.
2 Usage error.
3 No information is available about the specified terminal type.
4 The specified operand is invalid.
>4 An error occurred.

CONSEQUENCES OF ERRORS
If one of the operands is not available for the terminal, *tput* continues processing the remaining operands.

APPLICATION USAGE
The difference between resetting and initialising a terminal is left unspecified, as they vary greatly based on hardware types. In general, resetting is a more severe action.

Some terminals use control characters to perform the stated functions, and on such terminals it might make sense to use *tput* to store the initialisation strings in a file or environment variable for later use. However, because other terminals might rely on system calls to do this work, the standard output cannot be used in a portable manner, such as the following non-portable constructs:

```
ClearVar=‘tput clear‘
tput reset | mailx -s "Wake Up" ddg
```

EXAMPLES

1. Initialise the terminal according to the type of terminal in the environmental variable *TERM*. This command can be included in a **.profile** file.

   ```
   tput init
   ```

2. Reset a 450 terminal.

   ```
   tput -T 450 reset
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

stty, *tabs*.

CHANGE HISTORY

First released in Issue 4.

NAME

tr – translate characters

SYNOPSIS

```
tr [-cs] string1 string2
tr -s[-c] string1
tr -d[-c] string1
tr -ds[-c] string1 string2
```

DESCRIPTION

The *tr* utility copies the standard input to the standard output with substitution or deletion of selected characters. The options specified and the *string1* and *string2* operands control translations that occur while copying characters and single-character collating elements.

OPTIONS

The *tr* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c Complement the set of characters specified by *string1*. See **EXTENDED DESCRIPTION**.

–d Delete all occurrences of input characters that are specified by *string1*.

–s Replace instances of repeated characters with a single character, as described in **EXTENDED DESCRIPTION**.

OPERANDS

The following operands are supported:

string1
string2 Translation control strings. Each string represents a set of characters to be converted into an array of characters used for the translation. For a detailed description of how the strings are interpreted, see **EXTENDED DESCRIPTION**.

STDIN

The standard input can be any type of file.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tr*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of range expressions and equivalence classes.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- versus multi-byte characters in arguments) and the behaviour of character classes.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The *tr* output is identical to the input, with the exception of the specified transformations.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
The operands *string1* and *string2* (if specified) define two arrays of characters. The constructs in the following list can be used to specify characters or single-character collating elements. If any of the constructs result in multi-character collating elements, *tr* will exclude, without a diagnostic, those multi-character elements from the resulting array.

character
Any character not described by one of the conventions below represents itself.

octal Octal sequences can be used to represent characters with specific coded values. An octal sequence consists of a backslash followed by the longest sequence of one-, two- or three-octal-digit characters (01234567). The sequence causes the character whose encoding is represented by the one-, two- or three-digit octal integer to be placed into the array. If the size of a byte on the system is greater than nine bits, the valid escape sequence used to represent a byte is implementation-dependent. Multi-byte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.

character
The backslash-escape sequences in the table in the **XBD** specification, **Chapter 3**, **File Format Notation** (\\, \a, \b, \f, \n, \r, \t, \v) are supported. The results of using any other character, other than an octal digit, following the backslash are unspecified.

c–c Represents the range of collating elements between the range endpoints, inclusive, as defined by the current setting of the LC_COLLATE locale category. The starting endpoint must precede the second endpoint in the current collation order. The characters or collating elements in the range are placed in the array in ascending collation sequence.

[:*class*:] Represents all characters belonging to the defined character class, as defined by the current setting of the LC_CTYPE locale category. The following character class names will be accepted when specified in *string1*:

alnum	**blank**	**digit**	**lower**	**punct**	**upper**
alpha	**cntrl**	**graph**	**print**	**space**	**xdigit**

EX In addition, character class expressions of the form **[:***name***:]** are recognised in those locales where the *name* keyword has been given a **charclass** definition in the LC_CTYPE category.

When both the **–d** and **–s** options are specified, any of the character class names will be accepted in *string2*. Otherwise, only character class names **lower** or **upper** are valid in *string2* and then only if the corresponding character class (upper and **lower**, respectively) is specified in the same relative position in *string1*. Such a specification is interpreted as a request for case conversion. When [:lower:] appears in *string1* and [:upper:] appears in *string2*, the arrays will contain the characters from the **toupper** mapping in the LC_CTYPE category of the current locale. When [:upper:] appears in *string1* and [:lower:] appears in *string2*, the arrays will contain the characters from the **tolower** mapping in the LC_CTYPE category of the current locale. The first character from each mapping pair will be in the array for *string1* and the second character from each mapping pair will be in the array for *string2* in the same relative position.

Except for case conversion, the characters specified by a character class expression are placed in the array in an unspecified order.

If the name specified for *class* does not define a valid character class in the current locale, the behaviour is undefined.

[=*equiv*=]
Represents all characters or collating elements belonging to the same equivalence class as *equiv*, as defined by the current setting of the LC_COLLATE locale category. An equivalence class expression is allowed only in *string1*, or in *string2* when it is being used by the combined **–d** and **–s** options. The characters belonging to the equivalence class are placed in the array in an unspecified order.

[*x***n*] Represents *n* repeated occurrences of the character *x*. Because this expression is used to map multiple characters to one, it is only valid when it occurs in *string2*. If *n* is omitted or is zero, it is interpreted as large enough to extend the *string2*-based sequence to the length of the *string1*-based sequence. If *n* has a leading zero, it is interpreted as an octal value. Otherwise, it is interpreted as a decimal value.

When the **–d** option is not specified:

- Each input character found in the array specified by *string1* is replaced by the character in the same relative position in the array specified by *string2*. When the array specified by *string2* is shorter that the one specified by *string1*, the results are unspecified.
- If the **–c** option is specified, the complements of the characters specified by *string1* (the set of all characters in the current character set, as defined by the current setting of LC_CTYPE, except for those actually specified in the *string1* operand) are placed in the array in ascending collation sequence, as defined by the current setting of LC_COLLATE.
- Because the order in which characters specified by character class expressions or equivalence class expressions is undefined, such expressions should only be used if the intent is to map several characters into one. An exception is case conversion, as described previously.

When the **–d** option is specified:

- Input characters found in the array specified by *string1* will be deleted.
- When the **–c** option is specified with **–d**, all characters except those specified by *string1* will be deleted. The contents of *string2* will be ignored, unless the **–s** option is also specified.
- The same string cannot be used for both the **–d** and the **–s** option; when both options are specified, both *string1* (used for deletion) and *string2* (used for squeezing) are required.

When the **–s** option is specified, after any deletions or translations have taken place, repeated sequences of the same character will be replaced by one occurrence of the same character, if the character is found in the array specified by the last operand. If the last operand contains a character class, such as the following example:

```
tr -s '[:space:]'
```

the last operand's array will contain all of the characters in that character class. However, in a case conversion, as described previously, such as:

```
tr -s '[:upper:]' '[:lower:]'
```

the last operand's array will contain only those characters defined as the second characters in each of the **toupper** or **tolower** character pairs, as appropriate.

An empty string used for *string1* or *string2* produces undefined results.

EXIT STATUS

The following exit values are returned:

0 All input was processed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

If necessary, *string1* and *string2* can be quoted to avoid pattern matching by the shell.

If an ordinary digit (representing itself) is to follow an octal sequence, the octal sequence must use the full three digits to avoid ambiguity.

When *string2* is shorter than *string1*, a difference results between historical System V and BSD systems. A BSD system will pad *string2* with the last character found in *string2*. Thus, it is possible to do the following:

```
tr 0123456789 d
```

which would translate all digits to the letter d. Since this area is specifically unspecified in the document, both the BSD and System V behaviours are allowed, but a portable application cannot rely on the BSD behaviour. It would have to code the example in the following way:

```
tr 0123456789 '[d*]'
```

It should be noted that, despite similarities in appearance, the string operands used by *tr* are not regular expressions.

Unlike some previous versions, the Issue 4 *tr* correctly processes NUL characters in its input stream. NUL characters can be stripped by using *tr* **–d '\000'**.

EXAMPLES

1. The following example creates a list of all words in *file1* one per line in *file2*, where a word is taken to be a maximal string of letters.

   ```
   tr -cs "[:alpha:]" "[\n*]" <file1 >file2
   ```

2. The next example translates all lower-case characters in **file1** to upper-case and writes the results to standard output.

   ```
   tr "[:lower:]" "[:upper:]" <file1
   ```

   Note that the caveat expressed in the corresponding Issue 3 example is no longer in effect. This case conversion is now a special case that employs the **tolower** and **toupper** classifications, ensuring that proper mapping is accomplished (when the locale is correctly defined).

3. This example uses an equivalence class to identify accented variants of the base character e in **file1**, which are stripped of diacritical marks and written to **file2**.

   ```
   tr "[=e=]" e <file1 >file2
   ```

FUTURE DIRECTIONS

None.

SEE ALSO

sed.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

true – return true value

SYNOPSIS

```
true
```

DESCRIPTION

The *true* utility will return with exit code zero.

OPTIONS

None.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

None.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

None.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

Default.

APPLICATION USAGE

This utility is typically used in shell scripts, as shown in the **EXAMPLE**. The special built-in utility **:** is sometimes more efficient than *true*.

EXAMPLES

This command will be executed forever:

```
while true
do
        command
done
```

FUTURE DIRECTIONS

None.

SEE ALSO

false, Section 2.9 on page 45.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

tsort – topological sort

SYNOPSIS

EX `tsort [file]`

DESCRIPTION

The *tsort* utility writes to standard output a totally ordered list of items consistent with a partial ordering of items contained in the input.

The input consists of pairs of items (non-empty strings) separated by blanks. Pairs of different items indicate ordering. Pairs of identical items indicate presence, but not ordering.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of a text file to order. If no *file* operand is given, the standard input is used.

STDIN

The standard input is a text file that is used if no *file* operand is given.

INPUT FILES

The input file named by the *file* operand is a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tsort*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file consisting of the order list produced from the partially ordered input.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *LC_COLLATE* variable does not affect the actions of *tsort*. The output ordering is not lexicographic, but depends on the pairs of items given as input.

EXAMPLES

The command:

```
tsort <<EOF
a b c c d e
g g
f g e f
h h
EOF
```

produces the output:

```
a
b
c
d
e
f
g
h
```

FUTURE DIRECTIONS

None.

SEE ALSO

None.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Internationalised environment variable support mandated.

NAME

tty – return user's terminal name

SYNOPSIS

OB `tty -s`

DESCRIPTION

The *tty* utility writes to the standard output the name of the terminal that is open as standard input. The name that is used is equivalent to the string that would be returned by the **XSH** specification *ttyname*( ) function.

OPTIONS

The *tty* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

OB **–s** Do not write the terminal name. Only the exit status will be affected by this option. The terminal status will be determined as if the **XSH** specification *isatty*( ) function were used.

OPERANDS

None.

STDIN

While no input is read from standard input, standard input will be examined to determine whether or not it is a terminal, and, if so, to determine the name of the terminal.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *tty*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

OB If the **–s** option is specified, standard output will not be used. If the **–s** option is not specified and standard input is a terminal device, a pathname of the terminal as specified by the **XSH** specification *ttyname*( ) will be written in the following format:

```
"%s\n", <terminal name>
```

Otherwise, a message will be written indicating that standard input is not connected to a terminal. In the POSIX locale, the *tty* utility will use the format:

```
"not a tty\n"
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Standard input is a terminal.
1 Standard input is not a terminal.
>1 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

This utility checks the status of the file open as standard input against that of a system-defined set of files. It is possible that no match can be found, or that the match found need not be the same file as that which was opened for standard input (although they are the same device).

The **–s** option is useful only if the exit code is wanted. It does not rely on the ability to form a valid pathname. Portable applications should use *test* **–t 0**.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

The **XSH** specification description of *isatty*( ), *ttyname*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

type – write a description of command type

SYNOPSIS

EX `type` *`name...`*

DESCRIPTION

The *type* utility indicates how each argument would be interpreted if used as a command name.

OPTIONS

None.

OPERANDS

The following operand is supported:

name A name to be interpreted.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *type*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the location of *name*, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output of *type* contains information about each operand in an unspecified format. The information provided typically identifies the operand as a shell built-in, function, alias or keyword, and where applicable, may display the operand's pathname.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *type* must be aware of the contents of the current shell execution environment (such as the lists of commands, functions and built-ins processed by *hash*), it is always provided as a shell regular built-in. If it is called in a separate utility execution environment, such as one of the following:

```
nohup type writer
find . -type f | xargs type
```

it might not produce accurate results.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

command.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Relocated from the *sh* description to reflect its status as a regular built-in utility.

NAME

ulimit – set or report file size limit

SYNOPSIS

EX `ulimit [-f][`*`blocks`*`]`

DESCRIPTION

The *ulimit* utility sets or reports the file-size writing limit imposed on files written by the shell and its child processes (files of any size may be read). Only a process with appropriate privileges can increase the limit.

OPTIONS

The *ulimit* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–f Set (or report, if no *blocks* operand is present), the file size limit in blocks. The **–f** option is also the default case.

OPERANDS

The following operand is supported:

blocks The number of 512-byte blocks to use as the new file size limit.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *ulimit*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is used when no *blocks* operand is present. If the current number of blocks is limited, the number of blocks in the current limit is written in the following format:

```
"%d\n", <number of 512-byte blocks>
```

If there is no current limit on the number of blocks, in the POSIX locale the following format is used:

```
"unlimited\n"
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 A request for a higher limit was rejected or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *ulimit* affects the current shell execution environment, it is always provided as a shell regular built-in. If it is called in separate utility execution environment, such as one of the following:

```
nohup ulimit -f 10000
env ulimit 10000
```

it will not affect the file size limit of the caller's environment.

Once a limit has been decreased by a process, it cannot be increased (unless appropriate privileges are involved), even back to the original system limit.

EXAMPLES

Set the file size limit to 51,200 bytes:

```
ulimit -f 100
```

FUTURE DIRECTIONS

None.

SEE ALSO

The **XSH** specification description of *ulimit*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Relocated from the *sh* description to reflect its status as a regular built-in utility.

NAME

umask – get or set the file mode creation mask

SYNOPSIS

`umask [-S][`*`mask`*`]`

DESCRIPTION

The *umask* utility will set the file mode creation mask of the current shell execution environment (see Section 2.12 on page 63) to the value specified by the *mask* operand. This mask will affect the initial value of the file permission bits of subsequently created files. If *umask* is called in a subshell or separate utility execution environment, such as one of the following:

```
(umask 002)
nohup umask ...
find . -exec umask ... \;
```

it will not affect the file mode creation mask of the caller's environment.

If the *mask* operand is not specified, the *umask* utility will write to standard output the value of the invoking process's file mode creation mask.

OPTIONS

The *umask* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–S Produce symbolic output.

The default output style is unspecified, but will be recognised on a subsequent invocation of *umask* on the same system as a *mask* operand to restore the previous file mode creation mask.

OPERANDS

The following operand is supported:

mask A string specifying the new file mode creation mask. The string is treated in the same way as the *mode* operand described in the **EXTENDED DESCRIPTION** for *chmod*.

For a *symbolic_mode* value, the new value of the file mode creation mask will be the logical complement of the file permission bits portion of the file mode specified by the *symbolic_mode* string.

In a *symbolic_mode* value, the permissions *op* characters + and – will be interpreted relative to the current file mode creation mask; + will cause the bits for the indicated permissions to be cleared in the mask; – will cause the bits for the indicated permissions to be set in the mask.

The interpretation of *mode* values that specify file mode bits other than the file permission bits is unspecified.

EX In the obsolescent octal integer form of *mode*, the specified bits will be set in the file mode creation mask.

The file mode creation mask will be set to the resulting numeric value.

The default output of a prior invocation of *umask* on the same system with no operand will also be recognised as a *mask* operand. The use of an operand obtained in this way is not obsolescent, even if it is an octal number.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *umask*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

When the *mask* operand is not specified, the *umask* utility will write a message to standard output that can later be used as a *umask mask* operand.

If –**S** is specified, the message will be in the following format:

```
"u=%s,g=%s,o=%s\n", <owner permissions>, <group permissions>,
<other permissions>
```

where the three values will be combinations of letters from the set {r, w, x}; the presence of a letter will indicate that the corresponding bit is clear in the file mode creation mask.

If a *mask* operand is specified, there will be no output written to standard output.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The file mode creation mask was successfully changed, or no *mask* operand was supplied.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *umask* affects the current shell execution environment, it is generally provided as a shell regular built-in.

In contrast to the negative permission logic provided by the file mode creation mask and the octal number form of the *mask* argument, the symbolic form of the *mask* argument specifies those permissions that are left alone.

The references to octal modes are marked EX because, although they are obsolescent in the ISO/IEC 9945-2:1993 standard, X/Open systems have committed to maintaining them for portable applications until further notice.

EXAMPLES

Either of the commands:

```
umask a=rx,ug+w
```

```
umask 002
```

sets the mode mask so that subsequently created files have their S_IWOTH bit cleared.

After setting the mode mask with either of the above commands, the *umask* command can be used to write out the current value of the mode mask:

```
$ umask
0002
```

(The output format is unspecified, but historical implementations use the obsolescent octal integer mode format.)

```
$ umask -S
u=rwx,g=rwx,o=rx
```

Either of these outputs can be used as the mask operand to a subsequent invocation of the *umask* utility.

Assuming the mode mask is set as above, the command:

```
umask g-w
```

sets the mode mask so that subsequently created files have their S_IWGRP, and S_IWOTH bits cleared.

The command:

```
umask -- -w
```

sets the mode mask so that subsequently created files have all their write bits cleared. Note that *mask* operands **–r**, **–w**, **–x** or anything beginning with a hyphen, must be preceded by **––** to keep it from being interpreted as an option.

FUTURE DIRECTIONS

None.

SEE ALSO

chmod, the **XSH** specification description of *umask*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

unalias – remove alias definitions

SYNOPSIS

```
unalias alias-name...

unalias -a
```

DESCRIPTION

The *unalias* utility removes the definition for each alias name specified. See Section 2.3.1 on page 24. The aliases are removed from the current shell execution environment; see Section 2.12 on page 63.

OPTIONS

The *unalias* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–a Remove all alias definitions from the current shell execution environment.

OPERANDS

The following operand is supported:

alias-name
The name of an alias to be removed.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *unalias*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.

>0 One of the *alias-name* operands specified did not represent a valid alias definition, or an error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Since *unalias* affects the current shell execution environment, it is generally provided as a shell regular built-in.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

alias.

CHANGE HISTORY

First released in Issue 4.

NAME

uname – return system name

SYNOPSIS

```
uname [snrvma]
```

DESCRIPTION

By default, the *uname* utility will write the operating system name to standard output. When options are specified, symbols representing one or more system characteristics will be written to the standard output. The format and contents of the symbols are implementation-dependent. On systems conforming to the **XSH** specification, the symbols written will be those supported by the **XSH** specification *uname*( ) function.

OPTIONS

The *uname* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–a Behave as though all of the options **–mnrsv** were specified.

–m Write the name of the hardware type on which the system is running to standard output.

–n Write the name of this node within an implementation-specified communications network.

–r Write the current release level of the operating system implementation.

–s Write the name of the implementation of the operating system.

–v Write the current version level of this release of the operating system implementation.

If no options are specified, the *uname* utility will write the operating system name, as if the **–s** option had been specified.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uname*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
By default, the output will be a single line of the following form:

```
"%s\n", <sysname>
```

If the –**a** option is specified, the output will be a single line of the following form:

```
"%s %s %s %s %s\n", <sysname>, <nodename>, <release>, <version>,
<machine>
```

Additional implementation-dependent symbols may be written; all such symbols will be written at the end of the line of output before the newline character.

If options are specified to select different combinations of the symbols, only those symbols will be written, in the order shown above for the –**a** option. If a symbol is not selected for writing, its corresponding trailing blank characters also will not be written.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 The requested information was successfully written.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Note that any of the symbols could include embedded space characters, which may affect parsing algorithms if multiple options are selected for output.

The node name is typically a name that the system uses to identify itself for intersystem communication addressing.

EXAMPLES
The following command:

```
uname –sr
```

writes the operating system name and release level, separated by one or more blank characters.

FUTURE DIRECTIONS
None.

SEE ALSO

The **XSH** specification description of *uname*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

Issue 4, Version 2

The **SYNOPSIS** lists all the valid options.

NAME

uncompress – expand compressed data

SYNOPSIS

EX `uncompress [-cfv][`*`file...`*`]`

DESCRIPTION

The *uncompress* utility will restore files to their original state after they have been compressed using the *compress* utility. If no files are specified, the standard input will be uncompressed to the standard output. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file are preserved.

This utility supports the uncompressing of any files produced by the *compress* utility on the same implementation. For files produced by *compress* on other systems, *uncompress* supports 9- to 14-bit compression (see *compress* –**b**); it is implementation-dependent whether values of –**b** greater than 14 are supported.

OPTIONS

The *uncompress* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c Write to standard output; no files will be changed.

–f Do not prompt for overwriting files. Except when run in the background, if –**f** is not given the user will be prompted as to whether an existing file should be overwritten. If the standard input is not a terminal and –**f** is not given, *uncompress* will write a diagnostic message to standard error and exit with a status greater than zero.

–v Write messages to standard error concerning the expansion of each file.

OPERANDS

The following operand is supported:

file A pathname of a file. If *file* already has the .Z suffix specified, it will be used as the input file and the output file will be named *file* with the .Z suffix removed. Otherwise, *file* will be used as the name of the output file and *file* with the .Z suffix appended will be used as the input file.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –.

INPUT FILES

Input files are in the format produced by the *compress* utility.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uncompress*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
When there are no file operands or the **–c** option is specified, the uncompressed output will be written to standard output.

STDERR
Prompts will be written to the standard error output under the conditions specified in the **DESCRIPTION**. and **OPTIONS** sections. The prompts will contain the *file* pathname, but their format is otherwise unspecified. Otherwise, the standard error output will be used only for diagnostic messages.

OUTPUT FILES
Output files are the same as the respective input files to *compress*.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
The input file will remain unmodified.

APPLICATION USAGE
The limit of 14 on the *compress* **–b** *bits* argument is to achieve portability to all systems (within the restrictions imposed by the lack of an explicit published file format). Some systems based on 16-bit architectures cannot support 15- or 16-bit uncompression.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
compress, *unpack*, *zcat*.

CHANGE HISTORY
First released in Issue 4.

Issue 4, Version 2
The **DESCRIPTION** is clarified to state that the ownership, modes, access time, and modification time of the original file are preserved if the invoking process has appropriate privileges.

NAME

unexpand – convert spaces to tabs

SYNOPSIS

unexpand **[** -a**|** -t *tablist***][***file***..]**

DESCRIPTION

The *unexpand* utility copies files or standard input to standard output, converting blank characters at the beginning of each line into the maximum number of tab characters followed by the minimum number of space characters needed to fill the same column positions originally filled by the translated blank characters. By default, tabstops are set at every eighth column position. Each backspace character is copied to the output, and causes the column position count for tab calculations to be decremented; the count will never be decremented to a value less than one.

OPTIONS

The *unexpand* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following option is supported:

–a In addition to translating blank characters at the beginning of each line, translate all sequences of two or more blank characters immediately preceding a tab stop to the maximum number of tab characters followed by the minimum number of space characters needed to fill the same column positions originally filled by the translated blank characters.

–t *tablist*

Specify the tab stops. The option-argument *tablist* must be a single argument consisting of a single positive decimal integer or multiple positive decimal integers, separated by blank characters or commas, in ascending order. If a single number is given, tabs will be set *tablist* column positions apart instead of the default 8. If multiple numbers are given, the tabs will be set at those specific column positions.

Each tab-stop position *N* must be an integer value greater than zero, and the list must be in strictly ascending order. This is taken to mean that, from the start of a line of output, tabbing to position *N* will cause the next character output to be in the (*N*+1)th column position on that line. When the **–t** option is not specified, the default is the equivalent of specifying **–t 8** (except for the interaction with **–a**, described below).

No space-to-tab character conversions occur for characters at positions beyond the last of those specified in a multiple tab-stop list.

When **–t** is specified, the presence or absence of the **–a** option is ignored; conversion will not be limited to the processing of leading blank characters.

OPERANDS

The following operand is supported:

file A pathname of a text file to be used as input.

STDIN

See **INPUT FILES**.

INPUT FILES

The input files must be text files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *unexpand*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the processing of tab and space characters and for the determination of the width in column positions each character would occupy on a constant-width-font output device.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is equivalent to the input files with the specified space to tab character conversions.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

One non-intuitive aspect of *unexpand* is its restriction to leading spaces when neither **–a** nor **–t** is specified. Users who desire to always convert all spaces in a file can easily alias *unexpand* to use the **–a** or **–t 8** option.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

expand, *tabs*.

CHANGE HISTORY

First released in Issue 4.

NAME

unget – undo a previous get of an SCCS file (**DEVELOPMENT**)

SYNOPSIS

EX `unget [-ns][-r SID] file...`

DESCRIPTION

The *unget* utility reverses the effect of a *get* –**e** done prior to creating the intended new delta.

OPTIONS

The *unget* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–**r** *SID* Uniquely identify which delta is no longer intended. (This would have been specified by *get* as the new delta.) The use of this option is necessary only if two or more outstanding *get* commands for editing on the same SCCS file were done by the same person (login name).

–**s** Suppress the writing to standard output of the intended delta's SID.

–**n** Retain the file that was obtained by *get*, which would normally be removed from the current directory.

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file or a directory. If *file* is a directory, *unget* behaves as though each file in the directory were specified as a named file, except that non-SCCS files (last component of the pathname does not begin with s.) and unreadable files are silently ignored.

If a single instance *file* is specified as –, 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.

STDIN

The standard input is a text file used only when the *file* operand is specified as –. Each line of the text file is interpreted as an SCCS pathname.

INPUT FILES

Any SCCS files processed are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *unget*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output consists of a line for each file, in the following format:

```
"%s\n", <SID removed from file>
```

If there is more than one named file or if a directory or standard input is named, each pathname is written before each of the preceding lines:

```
"\n%s:\n", <pathname>
```

STDERR
Used only for diagnostic messages.

OUTPUT FILES
Any SCCS files updated are files of an unspecified format. During processing of a *file*, a locking *z-file*, as described in *get*, and a *q-file* (a working copy of the *p-file*), may be created and deleted. The *p-file* and *g-file*, as described in *get*, are deleted.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
None.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
delta, *get*, *sact*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

uniq – report or filter out repeated lines in a file

SYNOPSIS

```
uniq [-c|-d|-u][-f fields][-s char][input_file [output_file]]
```

OB

```
uniq [-c|-d|-u][-n][+m][input_file [output_file]]
```

DESCRIPTION

The *uniq* utility will read an input file comparing adjacent lines, and write one copy of each input line on the output. The second and succeeding copies of repeated adjacent input lines will not be written.

Repeated lines in the input will not be detected if they are not adjacent.

OPTIONS

OB The *uniq* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**; the obsolescent version does not, as one of the options begins with + and the –*m* and +*n* options do not have option letters.

The following options are supported:

–c Precede each output line with a count of the number of times the line occurred in the input.

–d Suppress the writing of lines that are not repeated in the input.

–f *fields*

Ignore the first *fields* fields on each input line when doing comparisons, where *fields* is a positive decimal integer. A field is the maximal string matched by the basic regular expression:

```
[[:blank:]]*[^[:blank:]]*
```

If the *fields* option-argument specifies more fields than appear on an input line, a null string will be used for comparison.

–s *chars*

Ignore the first *chars* characters when doing comparisons, where *chars* is a positive decimal integer. If specified in conjunction with the **–f** option, the first *chars* characters after the first *fields* fields will be ignored. If the *chars* option-argument specifies more characters than remain on an input line, a null string will be used for comparison.

–u Suppress the writing of lines that are repeated in the input.

OB **–***n* Equivalent to **–f** *fields* with *fields* set to *n*.

OB **+***m* Equivalent to **–s** *chars* with *chars* set to *m*.

OPERANDS

The following operands are supported:

input_file

A pathname of the input file. If the *input_file* operand is not specified, or if the *input_file* is –, the standard input will be used.

output_file

A pathname of the output file. If the *output_file* operand is not specified, the standard output will be used. The results are unspecified if the file named by *output_file* is the file named by *input_file*.

STDIN

The standard input will be used only if no *input_file* operand is specified or if *input_file* is –. See **INPUT FILES**.

INPUT FILES

The input file must be a text file.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uniq*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) which characters constitute a blank character in the current locale.

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output will be used only if no *output_file* operand is specified. See **OUTPUT FILES**.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

If the **–c** option is specified, the output file must be empty or each line must be of the form:

```
"%d %s", <number of duplicates>, <line>
```

otherwise, the output file must be empty or each line must be of the form:

```
"%s", <line>
```

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 The utility executed successfully.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *sort* utility can be used to cause repeated lines to be adjacent in the input file.

EXAMPLES

The following input file data (but flushed left) was used for a test series on *uniq*:

```
#01 foo0 bar0 foo1 bar1
#02 bar0 foo1 bar1 foo1
#03 foo0 bar0 foo1 bar1
#04
#05 foo0 bar0 foo1 bar1
#06 foo0 bar0 foo1 bar1
#07 bar0 foo1 bar1 foo0
```

What follows is a series of test invocations of the *uniq* utility that use a mixture of *uniq* options against the input file data. These tests verify the meaning of *adjacent*. The *uniq* utility views the input data as a sequence of strings delimited by \n. Accordingly, for the *fields*th member of the sequence, *uniq* interprets unique or repreated adjacent lines strictly relative to the *fields*+1th member.

1. This first example tests the line counting option, comparing each line of the input file data starting from the second field:

   ```
   uniq –c –f 1 uniq_0I.t
     1 #01 foo0 bar0 foo1 bar1
     1 #02 bar0 foo1 bar1 foo0
     1 #03 foo0 bar0 foo1 bar1
     1 #04
     2 #05 foo0 bar0 foo1 bar1
     1 #07 bar0 foo1 bar1 foo0
   ```

   The number 2, prefixing the fifth line of output, signifies that the *uniq* utility detected a pair of repeated lines. Given the input data, this can only be true when *uniq* is run using the **–f 1** option (which causes *uniq* to ignore the first field on each input line).

2. The second example tests the option to suppress unique lines, comparing each line of the input file data starting from the second field:

   ```
   uniq –d –f 1 uniq_0I.t
   #05 foo0 bar0 foo1 bar1
   ```

3. This test suppresses repeated lines, comparing each line of the input file data starting from the second field:

   ```
   uniq –u –f 1 uniq_0I.t
   #01 foo0 bar0 foo1 bar1
   #02 bar0 foo1 bar1 foo1
   #03 foo0 bar0 foo1 bar1
   #04
   #07 bar0 foo1 bar1 foo0
   ```

4. This suppresses unique lines, comparing each line of the input file data starting from the third character:

   ```
   uniq –d –s 2 uniq_0I.t
   ```

   In the last example, the *uniq* utility found no input matching the above criteria.

FUTURE DIRECTIONS

None.

SEE ALSO

comm, *sort*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

unpack – expand files (**TO BE WITHDRAWN**)

SYNOPSIS

EX `unpack` *`file...`*

DESCRIPTION

The *unpack* utility replaces files in the format used by *pack* with their unpacked form. For each file *file* operand, a search is made for a file called *file*.z (or just *file*, if *file* 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 stripped from its name. If the invoking process has appropriate privileges, the ownership, modes, access time, and modification time of the original file are preserved.

A file is not unpacked if one of the following is true:

- The filename (exclusive of the **.z**) has more than {NAME_MAX} bytes.
- The file cannot be opened.
- The file does not appear to be the output of *pack*.
- A file with the unpacked name already exists.
- The unpacked file cannot be created.

OPTIONS

None.

OPERANDS

The following operand is supported:

file A pathname of a file to be unpacked; *file* can include or omit the .z suffix.

STDIN

Not used.

INPUT FILES

The input files are regular files in the format created by *pack*.

ENVIRONMENT VARIABLES

The following environment variables may affect the execution of *pack*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If an error occurs, the $(basename file .z) file is not created and the original file is unchanged.

STDOUT

The standard output is a text file containing one line for each file unpacked, with the following format in the POSIX locale:

```
"unpack: %s: unpacked\n", file
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Files equivalent to the original unpacked file are created with the names as though $(basename file .z). were invoked corresponding to each *file* operand.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

This utility is being withdrawn from a future issue. The *uncompress* utility should be used instead.

SEE ALSO

pack, *pcat*, *uncompress*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Split into a separate description.

Marked **TO BE WITHDRAWN**.

Internationalised environment variable support made optional.

Issue 4, Version 2

The **DESCRIPTION** is revised as follows:

- It states that the ownership, modes, access time, and modification time of the original file are preserved if the invoking process has appropriate privileges.

- The assertion that a file is not unpacked if the filename has more than {NAME_MAX}–2 bytes now specifies {NAME_MAX} bytes.

NAME

uucp – system-to-system copy

SYNOPSIS

UN EX `uucp [-cCdfjmr][-n user] source-file... destination-file`

DESCRIPTION

The *uucp* utility copies files named by the *source-file* arguments to the *destination-file* argument. The files named can be on local or remote systems.

OPTIONS

The *uucp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–c Do not copy local file to the spool directory for transfer to the remote machine (default).

UN **–C** Force the copy of local files to the spool directory for transfer.

–d Make all necessary directories for the file copy (default).

UN **–f** Do not make intermediate directories for the file copy.

UN **–j** Write the job identification string to standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.

–m Send mail to the requester when the copy is completed.

UN **–n** *user*
Notify *user* on the remote system that a file was sent.

UN **–r** Do not start the file transfer; just queue the job.

OPERANDS

The following operands are supported:

destination-file
source-file
A pathname of a file to be copied to, or from, respectively. Either name can be a pathname on the local machine, or can have the form:

`system-name!pathname`

where *system-name* is taken from a list of system names that *uucp* knows about; see *uuname*. The destination *system-name* can also be a list of names such as:

`system-name!system-name!...!system-name!pathname`

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 pattern matching notation characters ?, * and [. . .] appearing in *pathname* will be expanded on the appropriate system.

Pathnames can be one of:

1. An absolute pathname.
2. A pathname preceded by *~user* where *user* 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 is to the public directory (called ''*PUBDIR*''; the actual location of *PUBDIR* is implementation-specific).

3. A pathname preceded by *~/destination* where *destination* is appended to *PUBDIR*.

   Note: This destination will be treated as a filename 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 files in that directory.

4. Anything else is prefixed by the current directory.

If the result is an erroneous pathname 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 read, write and execute permissions given by *uucp* are implementation-dependent.

STDIN

Not used.

INPUT FILES

The files to be copied are regular files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uucp*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE

Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within bracketed filename patterns.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes within bracketed filename patterns (for example, '[[:lower:]]*').

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME

Determine the format of date and time strings output by *uucp*.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files (which may be on other systems) are copies of the input files.

If the **–m** is used, mail files will be modified.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The domain of remotely accessible files can (and for obvious security reasons usually should) be severely restricted.

Note that the ! character in addresses has to be escaped when using *csh* as a command interpreter because of its history substitution syntax. For *ksh* and *sh* the escape is not necessary, but may be used.

Typical implementations of this utility require a communications line configured to use the the **XBD** specification, **Chapter 9**, **General Terminal Interface** interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility will write an error message describing the problem and exit with a non-zero exit status.

As noted above, shell metacharacters appearing in pathnames are expanded on the appropriate system. On an internationalised system, this is done under the control of local settings of *LC_COLLATE* and *LC_CTYPE*. Thus, care should be taken when using bracketed filename patterns, as collation and typing rules may vary from one system to another. Also be aware that certain types of expression (that is, equivalence classes, character classes and collating symbols) need not be supported on non-internationalised systems.

The *uucp* utility cannot guarantee support for all character encodings in all circumstances. For example, transmission data may be restricted to 7-bits by the underlying network, 8-bit data and filenames need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the Portable Filename Character Set be used for naming files.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *uuencode*, *uulog*, *uuname*, *uustat*, *uux*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Split into a separate description.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

Presence of the utility mandated, even on systems where no communications are available.

NAME

uudecode – decode a binary file

SYNOPSIS

uudecode **[***file***]**

DESCRIPTION

The *uudecode* utility reads a file or standard input if no file is specified, that includes data created by the *uuencode* utility. The *uudecode* utility scans the input file, searching for data compatible with the format specified in *uuencode* and attempts to create or overwrite the file described by the data. The pathname, file access permission bits and contents for the file to be produced are all contained in that data. The mode bits of the created file will be set from the file access permission bits contained in the data; that is, other attributes of the mode, including the file mode creation mask (see *umask*), will not affect the file being produced.

If the pathname of the file to be produced exists, and the user does not have write permission on that file, *uudecode* will terminate with an error. If the pathname of the file to be produced exists, and the user has write permission on that file, the existing file will be overwritten.

If the input data was produced by *uuencode* on a system with a different number of bits per byte than on the target system, the results of *uudecode* are unspecified.

OPTIONS

None.

OPERANDS

The following operand is supported:

file The pathname of a file containing the output of *uuencode*.

STDIN

See **INPUT FILES**.

INPUT FILES

The input files must be files containing the output of *uuencode*.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uudecode*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The output file will be in the same format as the file originally encoded by *uuencode*.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The user who is invoking *uudecode* must have write permission on the specified file.

The output of *uuencode* is essentially an encoded bit stream that is not cognizant of byte boundaries. It is possible that a 9-bit byte target machine can process input from an 8-bit source, if it is aware of the requirement, but the reverse is unlikely to be satisfying. Of course, the only data that is meaningful for such a transfer between architectures is generally character data.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
uuencode.

CHANGE HISTORY
First released in Issue 4.

NAME

uuencode – encode a binary file

SYNOPSIS

uuencode **[***file***]** *decode_pathname*

DESCRIPTION

The *uuencode* utility writes an encoded version of the named input file, or standard input if no *file* is specified, to standard output. The output is encoded using the algorithm described in **STDOUT** and includes the file access permission bits (in *chmod* octal or symbolic notation) of the input file and the *decode_pathname*, for re-creation of the file on another system that conforms to this document.

OPTIONS

None.

OPERANDS

The following operands are supported:

decode_pathname
: The pathname of the file into which the *uudecode* utility will place the decoded file. If there are characters in *decode_pathname* that are not in the portable filename character set the results are unspecified.

file
: A pathname of the file to be encoded.

STDIN

See **INPUT FILES**.

INPUT FILES

Input files can be files of any type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uuencode*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file (encoded in the character set of the current locale) that begins with the line:

```
"begin␣%s␣%s\n", <mode>, decode_pathname
```

and ends with the line:

```
end\n
```

In both cases, the lines have no preceding or trailing blank characters.

The algorithm that is used for lines in between **begin** and **end** takes three octets as input and writes four characters of output by splitting the input at six-bit intervals into four octets, containing data in the lower six bits only. These octets are converted to characters by adding a value of 0x20 to each octet, so that each octet is in the range 0x20–0x5f, and then it is assumed to represent a printable character in the ISO/IEC 646:1991 standard encoded character set. It then will be translated into the corresponding character codes for the codeset in use in the current locale. (For example, the octet 0x41, representing A, would be translated to A in the current codeset, such as 0xc1 if it were EBCDIC.)

Where the bits of two octets are combined, the least significant bits of the first octet are shifted left and combined with the most significant bits of the second octet shifted right. Thus the three octets A, B, C are converted into the four octets:

```
0x20 + (( A >> 2                    ) & 0x3F)
0x20 + (((A << 4) | ((B >> 4) & 0xF)) & 0x3F)
0x20 + (((B << 2) | ((C >> 6) & 0x3)) & 0x3F)
0x20 + (( C                         ) & 0x3F)
```

These octets are then translated into the local character set.

Each encoded line contains a length character, equal to the number of characters to be decoded plus 0x20 translated to the local character set as described above, followed by the encoded characters. The maximum number of octets to be encoded on each line is 45.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The file is expanded by 35 percent (each three octets become four, plus control information) causing it to take longer to transmit.

Since this utility is intended to create files to be used for data interchange between systems with possibly different codesets, and to represent binary data as a text file, the ISO/IEC 646:1991 standard was chosen for a midpoint in the algorithm as a known reference point. The output

from *uuencode* is a text file on the local system. If the output were in the ISO/IEC 646:1991 standard codeset, it might not be a text file (at least because the newline characters might not match), and the goal of creating a text file would be defeated. If this text file was then carried to another machine with the same codeset, it would be perfectly compatible with that system's *uudecode*. If it was transmitted over a mail system or sent to a machine with a different codeset, it is assumed that, as for every other text file, some translation mechanism would convert it (by the time it reached a user on the other system) into an appropriate codeset. This translation only makes sense from the local codeset, not if the file has been put into a ISO/IEC 646:1991 standard representation first. Similarly, files processed by *uuencode* can be placed in *pax* archives, intermixed with other text files in the same codeset.

The algorithm is described in terms of 8-bit quantities, or octets. Since no byte alignment is implied, it will encode data from machines with any number of bits per byte. However, unless that encoded data is then decoded on a machine with the same number of bits per byte, the output might not be useful.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *uudecode*.

CHANGE HISTORY

First released in Issue 4.

NAME

uulog – query system-to-system transaction log

SYNOPSIS

UN EX `uulog [-s system]`

DESCRIPTION

The *uulog* utility writes the status of *uucp* or *uux* transactions to standard output.

OPTIONS

The *uulog* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–s *system*
: Write information about file transfer work involving system *system*. By default, information is written about all systems.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uulog*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME
: Determine the format of date and time strings output by *uulog*.

NLSPATH
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ
: Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file indicating the status of file transfer work. The format is unspecified.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

uucp, *uuname*, *uustat*, *uux*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Split into a separate description.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

uuname – list names of other known uucp systems

SYNOPSIS

UN EX `uuname [-l]`

DESCRIPTION

The *uuname* utility writes the *uucp* names of known systems, one per line.

OPTIONS

The *uucp* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–l Write the local system name to standard output.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uuname*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is a text file listing names of systems, in the following format:

`"%s\n", <system name>`

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

uucp, *uulog*, *uustat*, *uux*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Split into a separate description.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

uupick – receive public system-to-system file copies

SYNOPSIS

UN EX `uupick [-s system]`

DESCRIPTION

The *uupick* utility can be used by a user to accept or reject the files transmitted to the user. Specifically, *uupick* searches the public directory (called ''*PUBDIR*''; the actual location of *PUBDIR* is implementation-specific) on the user's system for files sent to the user. For each entry (file or directory) found, the user is prompted for each file or directory. The *uupick* utility 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 an absolute pathname 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 Write the content of the file to standard output.

q Stop and exit.

<EOF> Same as q.

!*command*
Escape to the command interpreter to execute *command*.

***** Write a usage summary for the possible responses described here.

OPTIONS

The *uupick* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–s *system*
Process only files sent from *system*.

OPERANDS

None.

STDIN

Used to read the user's response to each file or directory prompt.

INPUT FILES

The files to be copied are regular files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uupick*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Prompts are written to standard output in an unspecified format. The prompt will contain at least the sending system name and the name of the subject file or directory.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
The output files are copies of the input files.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
There is no option (such as the *SHELL* variable) to specify a different command interpreter for use with !*command*.

Writing a file using p can cause problems on some terminals if the file is not a text file or contains control characters.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
uucp, *uuto*, *uustat*, *uux*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

uustat – uucp status inquiry and job control

SYNOPSIS

UN EX `uustat [ -q| -k jobid| -r jobid]`

EX `uustat [-s system][-u user]`

DESCRIPTION

The *uustat* utility displays the status of, or cancels, previously specified *uucp* requests, or provides general status on *uucp* connections to other systems.

When no options are given, *uustat* writes to standard output the status of all *uucp* requests issued by the current user.

OPTIONS

The *uustat* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

UN **–q** Write 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 invoking *uustat* unless that user has appropriate privileges.

UN **–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 program from deleting the job until the jobs modification time reaches the limit imposed by the program.

–s *system*
Write the status of all *uucp* requests for remote system *system*.

–u *user* Write the status of all *uucp* requests issued by *user*.

OPERANDS

None.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uustat*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

LC_TIME
Determine the format of date and time strings output by *uustat*.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

TZ Determine the timezone used with date and time strings.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output consists of information about each job selected, in an unspecified format. The information will include at least the job ID, the user ID or name and the remote system name.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
Typical implementations of this utility require a communications line configured to use the the **XBD** specification, **Chapter 9**, **General Terminal Interface** interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility will write an error message describing the problem and exit with a non-zero exit status.

EXAMPLES
None.

FUTURE DIRECTIONS
None.

SEE ALSO
uucp.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

Presence of the utility mandated, even on systems where no communications are available.

NAME

uuto – send public system-to-system file copies

SYNOPSIS

UN EX `uuto [mp] source-file... destination`

DESCRIPTION

The *uuto* utility sends *source-files* to *destination*. The *uuto* utility uses the *uucp* facility to send files, while it allows the local system to control the file access. A *source-file* name is a pathname on the user's machine.

The files (or subtrees if directories are specified) are sent to a public directory (called ''*PUBDIR*''; the actual location of *PUBDIR* is implementation-specific) 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.

OPTIONS

The *uuto* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following options are supported:

–m Send mail to the sender when the copy is complete.

–p Copy the source file into the spool directory before transmission.

OPERANDS

The following operands are supported:

destination

A string of the form:

system-name!*user*

where *system-name* is taken from a list of system names that *uucp* knows about; see *uuname*. The argument *user* is the login name of someone on the specified system. The destination *system-name* can also be a list of names such as:

system-name!*system-name*!...!*system-name*!*user*

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.

source-file

A pathname of a file on the local system to be copied to *destination*.

STDIN

Not used.

INPUT FILES

The files to be copied are regular files.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uuto*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error, and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

None.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output files (which may be on other systems) are copies of the input files.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

Typical implementations of this utility require a communications line configured to use the the **XBD** specification, **Chapter 9**, **General Terminal Interface** interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility will write an error message describing the problem and exit with a non-zero exit status.

The *uuto* utility cannot guarantee support for all character encodings in all circumstances. For example, transmission data may be restricted to 7-bits by the underlying network, 8-bit data and filenames need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991

standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the Portable Filename Character Set be used for naming files.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mailx, *uucp*, *uuencode*, *uupick*, *uustat*, *uux*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

Presence of the utility mandated, even on systems where no communications are available.

NAME

uux – remote command execution

SYNOPSIS

EX `uux [-np] command-string`

UN EX `uux [-jnpf7] command-string`

UN OB EX `uux [-][-jn] command-string`

DESCRIPTION

The *uux* utility will gather zero or more files from various systems, execute a shell pipeline (see Section 2.9 on page 45) on a specified system, and then send the standard output of the command to a file on a specified system. Only the first command of a pipeline can have a *system-name*! prefix. All other commands in the pipeline are executed on the system of the first command.

The following restrictions are applicable to the shell pipeline processed by *uux*:

- In gathering files from different systems, pathname expansion is not performed by *uux*. Thus, a request such as:

  ```
  uux "c89 remsys!~/*.c"
  ```

  would attempt to copy the file named literally ***.c** to the local system.
- The redirection operators **>>**, **<<**, **>|** and **>&** cannot be used.
- The reserved word **!** cannot be used at the head of the pipeline to modify the exit status.
- Alias substitution is not performed.

A filename can be specified as for *uucp*; it can be an absolute pathname, a pathname preceded by *~name* (which is replaced by the corresponding login directory), a pathname specified as *~/dest* (*dest* is prefixed by the public directory called *''PUBDIR''*; the actual location of *PUBDIR* is implementation-specific), or a simple filename (which is prefixed by *uux* with the current directory). See *uucp* for the details.

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 filenames (without path or machine reference) must be unique within the *uux* request.

The *uux* utility will attempt to get all files to the execution system. For files that are output files, the filename must be escaped using parentheses.

The remote system will notify the user by mail if the requested command on the remote system was disallowed or the files were not accessible. This notification can be turned off by the **–n** option.

OPTIONS

The *uux* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that – is supported as an option in the obsolescent version, rather than an operand. The following options are supported:

–p

OB **–** Make the standard input to *uux* the standard input to the *command-string*.

UN **–j** Write the job identification string to standard output. This job identification can be used by *uustat* to obtain the status or terminate a job.

–n Do not notify the user if the command fails.

OPERANDS

The following operands are supported:

command-string
A string made up of one or more arguments that are similar to normal command arguments, except that the command and any filenames can be prefixed by ''*system-name*!''. A null *system-name* is interpreted as the local system.

STDIN

The standard input is not used unless the – or **–p** option is specified; in those cases, the standard input is made the standard input of the *command-string*.

INPUT FILES

Input files are selected according to the contents of *command-string*.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *uux*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output is not used unless the **–j** option is specified; in that case, the job identification string is written to standard output in the following format:

```
"%s\n", <jobid>
```

STDERR

Used only for diagnostic messages.

OUTPUT FILES

Output files are created or written, or both, according to the contents of *command-string*.

If the **–n** is not used, mail files will be modified following any command or file-access failures on the remote system.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

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*.

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.

Typical implementations of this utility require a communications line configured to use the the **XBD** specification, **Chapter 9**, **General Terminal Interface** interface, but other communications means may be used. On systems where there are no available communications means (either temporarily or permanently), this utility will write an error message describing the problem and exit with a non-zero exit status.

As noted in *uucp*, shell pattern matching notation characters appearing in pathnames are expanded on the appropriate local system. This is done under the control of local settings of *LC_COLLATE* and *LC_CTYPE*. Thus, care should be taken when using bracketed filename patterns, as collation and typing rules may vary from one system to another. Also be aware that certain types of expression (that is, equivalence classes, character classes and collating symbols) need not be supported on non-internationalised systems.

The *uux* utility cannot guarantee support for all character encodings in all circumstances. For example, transmission data may be restricted to 7-bits by the underlying network, 8-bit data and filenames need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used and that only characters defined in the Portable Filename Character Set be used for naming files.

EXAMPLES

1. The following command gets *file1* from system **a** and *file2* file from system **b**, executes *diff* on the local system, and puts the results in **file.diff** in the local *PUBDIR* directory. (*PUBDIR* is the *uucp* public directory on the local system.)

```
uux "!diff a!/usr/file1 b!/a4/file2 >!~/file.diff"
```

2. The following command will fail because *uux* places all files copied to a system in the same working directory. Although the files **xyz** are from two different systems, their filenames are the same and will conflict.

```
uux "!diff a!/usr1/xyz b!/usr2/xyz >!~/xyz.diff"
```

3. The following command will succeed (assuming *diff* is permitted on system **a**) because the file local to system **a** is not copied to the working directory, and hence does not conflict the file from system **c**.

```
uux "a!diff a!/usr/xyz c!/usr/xyz >!~/xyz.diff"
```

FUTURE DIRECTIONS

A version of *uux* that fully supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** may be introduced in a future issue.

SEE ALSO

uucp, *uuencode*, *uustat*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

Presence of the utility mandated, even on systems where no communications are available.

NAME

val – validate SCCS files (**DEVELOPMENT**)

SYNOPSIS

EX `val -`

EX `val [-s][-m name][-r SID][-y type] file...`

DESCRIPTION

The *val* utility determines if the specified *file* is an SCCS file meeting the characteristics specified by the options.

OPTIONS

The *val* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the usage of the – operand is not strictly as intended by the guidelines (that is, reading options and operands from standard input). The following options are supported:

–s Silence the diagnostic message normally written to standard output for any error that is detected while processing each named file on a given command line.

–r *SID* Specify a *SID* (SCCS Identification String), an SCCS delta number. A check is made to determine if the *SID* is ambiguous (for example, **–r 1** is ambiguous because it physically does not exist but implies 1.1, 1.2, and so forth, which may exist) or invalid (for example, **–r 1.0** or **–r 1.1.0** 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*
Specify a *name*, which is compared with the SCCS **%M%** keyword in *file*. (See *get*).

–y *type* Specify a *type*, which is compared with the SCCS **%Y%** keyword in *file*. (See *get*).

OPERANDS

The following operands are supported:

file A pathname of an existing SCCS file. If a single instance *file* is specified as –, and if no options are specified, the standard input is read: each line is independently processed as if it were a command-line argument list. (However, the line is not subjected to any of the shell word expansions, such as parameter expansion or quote removal.)

STDIN

The standard input is a text file used only when the *file* operand is specified as –.

INPUT FILES

Any SCCS files processed are files of an unspecified format.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *val*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error. and informative messages written to standard output.

NLSPATH
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
The standard output consists of informative messages about either: (1) each file processed, or; (2) each command line read from standard input.

If the standard input is not used, for each *file* operand yielding a discrepancy, the output line has the following format:

```
" %s: %s\n", <pathname>, <unspecified string>
```

If standard input is used, a line of input is written before each of the preceding lines for files containing discrepancies:

```
"%s:\n", <input line>
```

STDERR
Not used.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The 8-bit code returned by *val* is a disjunction of the possible errors, that is, it can be interpreted as a bit string where set bits are interpreted as follows:

0x80 = Missing file argument.
0x40 = Unknown or duplicate option.
0x20 = Corrupted SCCS file.
0x10 = Cannot open file or file not SCCS.
0x08 = *SID* is invalid or ambiguous.
0x04 = *SID* does not exist.
0x02 = **%Y1%**, **–y** mismatch.
0x01 = **%M%**, **–m** mismatch.

Note that *val* can process two or more files on a given command line and 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.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE

Since the *val* exit status sets the 0x80 bit, shell applications checking $? cannot tell if it terminated due to a missing file argument or receipt of a signal.

EXAMPLES

In a directory with three SCCS files, **s.x** (of **t** type ‘‘text’’), **s.y** and **s.z** (a corrupted file), the following command could produce the output shown:

```
val – <<EOF
–y source s.x
–m y s.y
s.z
EOF

–y source s.x

    s.x: %Y%, –y mismatch
s.z

    s.z: corrupted SCCS file
```

FUTURE DIRECTIONS

None.

SEE ALSO

admin, *delta*, *get*, *prs*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Exceptions to Utility Syntax Guidelines conformance noted.

Internationalised environment variable support mandated.

NAME

vi – screen-oriented (visual) display editor

SYNOPSIS

EX `vi [-rR][-l][-c command][-t tagstring][-w size][file...]`

EX OB `vi [-rR][-l][+command][-t tagstring][-w tagstring7][-w size][file...]`

DESCRIPTION

The *vi* (visual) utility is a screen-oriented text editor. The user can switch back and forth between *vi* and the line editor *ex* and execute *ex* commands from within *vi*.

When using *vi*, the terminal screen acts as a window into the editing buffer. Changes made to the editing buffer are reflected in the screen display; the position of the cursor on the screen indicates the position within the editing buffer.

Certain block-mode terminals do not have all the capabilities necessary to support the complete *vi* definition. When these commands cannot be supported on such terminals, this condition will not produce an error message such as ''not an editor command'' or report a syntax error. The implementation may either accept the commands and produce results on the screen that are the result of an unsuccessful attempt to meet the requirements of this document or report an error describing the terminal-related deficiency.

OPTIONS

OB The *vi* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines** except for the obsolescent +*command* option.

The following options are supported:

–c *command*

OB +*command*

Begin editing by executing the specified *ex* command-mode commands. As with normal *ex* command-line entries, the *command* option-argument can consist of multiple *ex* commands separated by vertical-line characters (|). The use of commands that enter input mode in this manner produces undefined results.

EX **–l** Set lisp mode (see **Edit Options in ex** on page 317).

–r Attempt to recover the named *file*s after an editor or system crash, after the editor has been terminated by a signal or after the use of a **pre** (*ex*) editor command.

If no *file* operands are given, all other options, the *EXINIT* variable, and any **.exrc** files will be ignored; a list of all recoverable files available to the invoking user will be written; and *vi* will exit without reading files or processing user commands.

–R Set read-only mode, preventing accidental overwriting of the files. Any command that would write to a file requires the ! suffix (see, for example, the **write** command) to be effective in this mode.

–t *tagstring*

Edit the file containing the specified *tagstring* and set the initial position within the edit buffer to the point of definition of the tag. (See *ctags*.) The tags feature represented by **–t** *tagstring* and the **ta** command is optional. It is provided on any system that also provides a conforming implementation of *ctags*; otherwise, the use of **–t** produces undefined results.

–w *size* Set the value of the *window* editor option to *size*. See the **window** option.

OB If both the **–t** *tagstring* and **–c** *command* (or the obsolescent **+***command*) options are given, the **–t** *tagstring* will be processed first; that is, the file containing the tag is selected by **–t** and then the command is executed.

OPERANDS

The following operand is supported:

file A pathname of a file to be edited.

STDIN

The standard input consists of a series of commands and input text, as described in **EXTENDED DESCRIPTION**.

INPUT FILES

Input files must be text files or files that would be text files except for an incomplete last line that is not longer than {LINE_MAX} – 1 bytes in length and contains no NUL characters. The editing of other forms of files may optionally be allowed by *vi* implementations.

The **.exrc** files (see **EXTENDED DESCRIPTION**) must be text files consisting of *ex* commands.

By default, *vi* will read lines from the files to be edited without interpreting any of those lines as any form of editor command.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *vi*:

COLUMNS
Override the system-selected horizontal screen size. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

EXINIT Determine a list of *ex* commands that will be executed on editor startup, before reading the first file. The list can contain multiple commands by separating them using a vertical-line (|) character. See also **EXTENDED DESCRIPTION** for more details of the initialisation phase.

HOME Determine a pathname of a directory that will be searched for an editor startup file named **.exrc**; see **EXTENDED DESCRIPTION**.

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements within regular expressions.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files), the behaviour of character classes within regular expressions, the classification of characters as upper- or lower-case letters, the case conversion of letters, and the detection of word boundaries.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

LINES Override the system-selected vertical screen size, used as the number of lines in a screenful and the vertical screen size in visual mode. See the **XBD** specification, **Chapter 6**, **Environment Variables** for valid values and results when it is unset or null.

PATH Determine the search path for the shell command specified in the editor commands **shell**, **read** and **write** and the visual-mode command !; see the description of command search and execution in **Command Search and Execution** on page 47.

SHELL Determine the preferred command-line interpreter for use in !, **shell**, **read** and other commands with an operand of the form !*string*. For the **shell** command the program will be invoked with the single argument **–i**, for all others it will be invoked with the two arguments **–c** and *string*. If this variable is null or not set, the *sh* utility will be used.

TERM Determine the name of the terminal type. If this variable is unset or null, an unspecified default terminal type will be used.

ASYNCHRONOUS EVENTS

The following actions will be taken upon receipt of signals:

SIGINT The current editor command will be aborted. The editor will prompt for another command.

SIGCONT
The screen will be refreshed (if in *visual* mode).

SIGHUP
If the current buffer has changed since the last e or w command, *vi* will attempt to save the current file in a state such that it can be recovered later by an *ex* or *vi* **–r** command.

The action taken for all other signals is unspecified.

STDOUT

If standard output is a terminal device, it may be used for writing prompts to the user, for informational messages and for writing lines from the file. If standard output is not a terminal device, undefined results occur.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The output from *vi* are text files that are identical to the input files if no changes have been made to those files by commands, with the exception that in all cases where a forced session termination (the *ex* command q!) has not been issued prior to any write of the file, a trailing newline character will be added to the last line of the file if one was not present in the input.

EXTENDED DESCRIPTION

Only the visual mode of the editor is described in this section. See the *ex* utility for additional editing capabilities used in *vi*.

The pathname of the file being edited by *vi* is the *current* file. The text of the file will be read into a *buffer*, and all editing changes will be performed in this buffer; changes will have no effect on

the file until the buffer is written out explicitly. Lines in the buffer may each be limited to {LINE_MAX} bytes and an error message may be displayed if the limit is exceeded during editing.

The *alternative* pathname is the name of the last file mentioned in an editor command, or the previous current pathname if the last file mentioned became the current file. When the character % appears in a pathname entered as part of a command argument, the character will be replaced by the current pathname; the character # will be replaced by the alternative pathname. The characters % and # can be escaped by preceding them with a backslash.

During initialisation, before the first file is read or any user commands from the terminal are processed, if the environment variable *EXINIT* is set, the editor will execute the *ex* commands contained in that variable. If the variable is not set, *vi* will attempt to read *ex* commands from the file **$HOME/.exrc** (the file **.exrc** in the directory referred to by the *HOME* environment variable). If and only if *EXINIT* or **$HOME/.exrc** sets the editor option **exrc**, *vi* finally will attempt to read *ex* commands from a file **.exrc** in the current directory. In the event that *EXINIT* is not set and the current directory is the user's home directory, any **.exrc** file will only be processed once. No **.exrc** file will be read unless it is owned by the same user ID as the effective user ID of the process. After any **.exrc** files are processed, any commands specified by the **–c** option will be processed.

After initialisation, *vi* will be in *command* mode; *input* mode can be entered by several commands used to insert or change text. In input mode, an escape character can be used to return to command mode; other uses of the escape character are described in **Terminate Command or Input Mode** on page 784. The results of entering newline characters in input mode are affected by the setting of the **autoindent** editor variable.

The last (bottom) line of the screen is used to display the input for search commands (**/** and **?**), for *ex* commands (**:**), and system commands (**!**). It is also used to report errors or display informational messages. The editor prompts for input for commands by displaying the command (/, ?, : or !) at the beginning of the last line of the screen. All subsequent characters will be then taken as input until a line terminator is entered.

An interrupt typed during text input, or during the input of a command on the bottom line, will terminate the input (or cancel the command) and will return the editor to command mode. During command mode, an interrupt will alert the terminal. In general, an alert indicates an error (such as unrecognised key).

Lines displayed on the screen containing only a tilde (~) indicate that the last line above them is the last line in the editing buffer (the ~ lines are past the end of the editing buffer).

There may be lines on the screen marked with an @. These indicate space on the screen not corresponding to lines in the file. This will happen if there is space at the bottom of the screen for some text, but the next line of the buffer will not fit. (It may also happen on a terminal with limited local intelligence. In this case, these lines can be removed by entering a <control>-R, forcing the editor to refresh the screen.)

Command Descriptions in vi

The cursor, in general, is placed on the current line and in the current column position as noted for each command described below. If the current line is not in the display window, then the display window will be either scrolled or refreshed to cause the current line to be in the display window. If the screen is refreshed, the current line will be positioned as close to the center of the display window as possible. If the current line is less than one-half window lines from the beginning of the editing buffer, the first line of the buffer will be displayed on the first line of the display window. If the current line is less than one-half window lines from the end of the

editing buffer, the remaining lines of the display window after the last line of the file will contain only a ~ character in column position 1. If the screen is scrolled rather than refreshed, the current line will be placed at the top of the display window if the current line is before the first line displayed. If the current line is after the last line displayed, the display window will be scrolled up and the current line will be placed on the last line of the display window.

As stated previously, the cursor is generally placed on the current column position of the current line. The one exception to this is if the current column position is beyond the end of the current line. In this case, the cursor will be placed on the last character of the current line. However, the current column position value will not be altered by this. Thus, if the current line changes to a longer line, the cursor will be moved back out to the current column position or to the end of the line if it is shorter than the current column.

Unless otherwise specified, the commands are interpreted in command mode and have no special effect in input mode.

Some of the following command descriptions (such as a, A, c, and so forth) move the cursor and enter input mode. The indications of *Current line* and *Current column* are for the cursor movement up to the beginning of input mode. Depending on the text inserted, the cursor will generally wind up in a totally different location by the time command mode is reentered. The cursor movement that accompanies the transition from input mode to command mode is described in **Terminate Command or Input Mode** on page 784.

The following symbols are used in this section to represent arguments to commands, to describe commands or to specify the new values of the current line and column indicators following execution of the command.

bigword A maximal sequence of non-blank characters preceded and followed by blank characters or the beginning or end of a line or the file.

buffer One of a number of named areas for saving text. Commands that change or delete text can be preceded by a buffer specification argument *buffer*. It is specified in the form "*c*, where *c* represents the name of a buffer, one of the lower-case letters of the POSIX locale. Specifying *buffer* will cause the area of text affected by the command to be stored into the buffer as it was before the command took effect. This argument is also used on the put commands (p and P) to specify the buffer that will provide the text to insert. When a command synopsis shows both **[***buffer***]** and **[***count***]** preceding the command letter, either can precede the other.

If the buffer name is specified in upper-case, and the buffer is to be modified (as with a deletion or yanking command) the buffer will be appended to rather than being overwritten. If the buffer is not to be modified (as in a visual mode put commands) the buffer name can be specified in lower-case or upper-case with the same results. There will be also one unnamed buffer, which is the repository for all text deleted (with the **delete** or visual mode d command) or yanked (with the **yank** or visual mode y command) when no buffer is specified.

There are also numbered buffers, 1 to 9, inclusive, which are accessible only from visual mode. These buffers are special in that, in visual mode, when deleted text is placed in the unnamed buffer, it also will be placed in buffer 1, the previous contents of buffer 1 will be placed in buffer 2, and so on. Any text in buffer 9 will be lost. Text that is yanked (or otherwise copied) into the unnamed buffer will not modify the numbered buffers. Text cannot be placed directly into the numbered buffers although it can be retrieved from them by using a visual mode put command with the buffer name given as a number. When the *buffer* modifier is not used in the commands below, the unnamed buffer is the default.

column The (previous) value of the current column indicator.

count A positive integer used as an optional argument to most commands, either to give a size or a position (for display or movement commands), or as a repeat count (for commands that change text). This argument is optional and defaults to 1 unless otherwise noted in the individual command description.

end-of-line
A description of the current column indicator value meaning that the current column indicator will be set always to indicate the last character of the current line. Until the current column indicator is changed, the cursor will be always positioned to the last character of whatever line is current.

line The (previous) value of the current line indicator.

non-blank
A description of the current column indicator value meaning that the current column indicator will be set to the position of the first non-blank character of the current line. If the current line has no non-blank characters, the indicator will be set to the position of the last character of the line.

motion A command used as an optional trailing argument to some commands, indicating the extent of text to be affected by the command. The *motion* argument can be either the command character repeated or a cursor movement command. In the former case, exactly the current line is affected. In the latter case, the region specified will be from the current cursor position to just before the cursor position indicated by the motion command. If the command operates on lines only, then all the lines that fall partly or wholly within this region are affected. Otherwise, the exact region as described above is affected. The following commands are considered to be cursor motion commands:

$	^	e	M
%	‘‘	<control>-F	<control>-N
’’	‘*letter*	F	N
’*letter*	{	f	n
(	|	G	<control>-P
)	}	<control>-H	<space>
,	0	H	<control>-T
-	<control>-B	<control>-J	T
/	B	j	t
;	b	k	<control>-U
?	<control>-D	L	W
[[	<control>-E	l	w
]]	E	<control>-M	<control>-Y

The optional *count* prefix available for some of the motion commands can be included and is considered part of the *motion* argument. For example, in **c2w**, the **2w** is the *motion* argument.

previous context
A state set whenever a non-relative motion command is executed in *vi*.

paragraph
An area of text that begins with either the beginning of a file, an empty line, or section boundary and continues until either an empty line, section boundary or the end of the editing buffer. Additional paragraph boundaries can be defined by the *ex* **paragraph** option.

section An area of text that starts with a line whose first character is either a form-feed character or an open brace ({) and continues until the next section or the end of the editing buffer. The beginning and end of the editing buffer are also considered section boundaries. Additional section boundaries can be defined by the *ex* **sections** option.

sentence
An area of text that begins with either the beginning of the editing buffer or the first non-blank character following the previous sentence, paragraph or section boundary and continues until the end of the editing buffer or a period, exclamation point or question mark (., !, ?) character followed by either an end of line or two space characters. Any number of closing parentheses, brackets or double-quote (),], ") characters can appear between the period, exclamation point or question mark and the space characters or end of line.

window The current value of the editor option **window**.

word In the POSIX locale, *vi* recognises two kinds of words:

- A maximal sequence of letters, digits and underscores, delimited at both ends by: characters other than letters, digits or underscores; the beginning or end of a line; or the end of the editing buffer.
- A maximal sequence of characters other than letters, digits, underscores or white space, delimited at both ends by: a letter, digit, underscore or white space; the beginning or end of a line; or the end of the editing buffer.

! A character that can be appended to the command to modify its operation as detailed in the individual command descriptions.

After processing a command, if the current line and column indicators have been changed to specify a character not currently on the screen, the editor will scroll or page the text to bring the needed character onto the screen. If the new current line is within *window* lines of the previous current line, and the terminal is capable of scrolling in the indicated direction, the text will be scrolled the minimum number of lines necessary to present the needed character. Otherwise, the screen will be redisplayed with the new current line positioned in the middle of the screen.

The following rules specify how exceptions will be handled. When a movement command would cause a window to display beyond the beginning of the editing buffer, the window displayed is the window beginning with the first line of the editing buffer. When a movement command would cause a window to be displayed beyond the end of the editing buffer, the terminal will be alerted and the command aborted. When the current line is empty, the cursor will be placed in the first position of the line on the screen. When the current column indicator value is beyond the end of the current line, the cursor will be placed on the last character of the current line, but the current column indicator will be unchanged.

The following rules specify how position exceptions will be handled. When the current column indicator is at the beginning (end) of a line and would be set to a position before the beginning (after the end) of the current line, the command will not be executed, the terminal will be alerted, and the current position will remain unchanged. Otherwise, when the current column indicator would be set to a position before the beginning (after the end) of the current line, it will be set instead to the position of the first (last) character of that line. When the current line indicator would be set to a position before the beginning (after the end) of the editing buffer, the command will not be executed, the terminal will be alerted, and the current position will remain unchanged.

Page Backwards

Synopsis: **[***count***]** <control>-B

Page backward in the file, allowing two lines of overlap, by displaying the window starting at line [*line* – (*count* * (*window*–2))].

Current line: The last line displayed.

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Scroll Forward

Synopsis: **[***count***]** <control>-D

Scroll forward *count* lines in the file. If *count* is not specified, the same number of lines will be scrolled as by the previous <control>-D or <control>-U command. On the first <control>-D or <control>-U command, the amount scrolled will be one-half the number of lines in a screenful. See the *ex* **scroll** option.

Current line: (*line* + amount scrolled).

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Scroll Forward by Line

Synopsis: **[***count***]** <control>-E

Scroll forward *count* lines, leaving the current line and column as is if possible.

Current line: Unchanged unless the current line scrolls off the screen; otherwise, move to the first line displayed.

Current column: Unchanged unless the current line scrolls off the screen; otherwise, move to the first character if it is a blank line or the last character of the line if the position of the current column is further into the line than the position of the last column of the line.

Page Forward

Synopsis: **[***count***]** <control>-F

Page forward in the file, allowing two lines of overlap, by displaying the window starting at line [*line* + (*count* ∗ (*window*–2))].

Current line: The first line displayed.

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Display Information

Synopsis: <control>-G

Display an informational message listing the current pathname, current line, number of lines and other unspecified information, as does the *ex* command *file*.

Current line: Unchanged.

Current column: Unchanged.

Move Cursor Backwards

Synopsis: **[***count***]** <control>-H
Synopsis: **[***count***]** h

Move the cursor back *count* characters on the current line. No characters will be erased from the screen by this command. In input mode, <control>-H will have the exact same function.

Current line: Unchanged.

Current column: Set to (*column* – the number of columns occupied by *count* characters ending with the previous current column) or the beginning of the line if *count* is greater than the number of characters preceding the current character in the current line.

Move Down in Column

Synopsis: **[***count***]** <control>-J
Synopsis: **[***count***]** j
Synopsis: **[***count***]** <control>-N

Move the cursor down *count* lines without changing the current column.

Current line: *current line* + *count*. Unchanged if *count* is greater than the number of lines in the buffer following the current line.

Current column: Unchanged.

Clear and Redisplay

Synopsis: <control>-L

Clear and redisplay the screen.

Current line: Unchanged.

Current column: Unchanged.

Move Cursor Down to Non-blank

Synopsis: **[***count***]** <control>-M
Synopsis: **[***count***]** +

Move the cursor down *count* lines to the first non-blank character of that line.

Current line: *line* + *count*.

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Move Up in Column

Synopsis: **[***count***]** <control>-P
Synopsis: **[***count***]** k

Move the cursor up *count* lines without changing the current column.

Current line: *line* – *count*.

Current column: Unchanged.

Redraw Screen

Synopsis: `<control>-R`

Redraw the current screen. If any lines have been deleted from the logical screen and flagged as deleted on the terminal using the @ convention (see the beginning of **EXTENDED DESCRIPTION**), they will be deleted, and the logical screen will be redisplayed. Lines flagged with @ because they do not fit on the terminal display will not be affected.

Current line: Unchanged.

Current column: Unchanged.

Move Forward with Tabs

Synopsis: `<control>-T`

Move the cursor forward *shiftwidth* (see the *ex* **shiftwidth** option) positions in insert mode by inserting tab characters, followed by space characters, as necessary, if the current cursor position is at the beginning of a line or preceded only by blank characters.

Current line: Unchanged.

Current column: *column* + *shiftwidth*.

Scroll Backward

Synopsis: `[`*count*`] <control>-U`

Scroll backward *count* lines in the file. If *count* is not specified, the same number of lines will be scrolled as by the previous <control>-D or <control>-U command. On the first <control>-D or <control>-U command, the amount scrolled will be the value of the *ex* **scroll** editor option.

Current line: (*line* – amount scrolled).

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Escape Next Character

Synopsis: `<control>-V` *character*
Synopsis: `<control>-Q` *character*

Allow the entry of a subsequent *character*, removing any special meaning to the editor it has in input mode (for example, the escape character). The character ^ will be displayed in the current cursor position until the subsequent character is typed, which then replaces the character ^.

Current line: Unchanged.

Current column: Unchanged.

Delete Word

Synopsis: `<control>-W`

Delete the word preceding the cursor (including any blank characters between the end of the word and the current cursor position) in input mode by moving the cursor back to the beginning of the word. No characters will be erased from the screen by this command. Only text entered during the current input mode session and on the current line can be deleted with this command.

Current line: Unchanged.

Current column: Moved back as described above.

Scroll Backward by Line

Synopsis: **[***count***]** <control>-Y

Scroll backward *count* lines, leaving the current line and column as is, if possible.

Current line: Unchanged unless the current line scrolls off the screen; otherwise, the last line displayed.

Current column: Unchanged unless the current line scrolls off the screen; otherwise, move to the first character if it is a blank line or the last character of the line if the position of the current column is further into the line than the position of the last column of the line.

Terminate Command or Input Mode

Synopsis: <ESC>

The effects of the escape character depend on the mode and the command being entered, as follows:

- In command mode:
  - — If a line-oriented command (/, ?, :, or !) is being entered, terminate the line-oriented command and execute it.
  - — If only part of a command has been entered, cancel the partial command. A command is not considered to be partially entered until at least one non-*count* character has been entered. For example, 33c<ESC> is partially entered and silently canceled, whereas 33<ESC> alerts the terminal (see the next item).
  - — Otherwise, alert the terminal.
- In input mode: Terminate input mode and return to command mode. At this point any text that was deleted in input mode, but that has not yet been erased from the screen, will be erased from the screen; the current line will be redisplayed to match exactly the changes made in input mode.

Current line: When a line-oriented command is being terminated, the current line indicator will be set as given for that command. Otherwise, unchanged.

Current column: When a line-oriented command is being terminated, the current column indicator will be set as given for that command. When terminating input mode, if the cursor is not at the beginning of a line, the current column position will be set to (*column* – the width of the last character inserted). Otherwise, unchanged.

Move Cursor Forward

Synopsis: **[***count***]** <space>
Synopsis: **[***count***]** l (ell)

Move the cursor forward *count* characters without changing the current line.

Current line: Unchanged.

Current column: (*current column* + the width of the next *count* characters) or the end of the line if *count* is greater than the number of characters in the line following the current column.

Replace Text with Results from Shell Command

Synopsis: **[***count***]** ! *motion shell-commands* <newline>

Pass the lines specified by *count* and *motion* as standard input to a shell command, for the program named in the *SHELL* environment variable, and replace those lines with the standard output of the shell command. An implementation may consider a *motion* command that does not result in an integral number of lines to be an error. After the *motion* is entered, the text of the command will be prompted for on the last line of the display as described at the beginning of **EXTENDED DESCRIPTION**. A warning will be issued if the buffer has been changed since the last write.

Within the text of *command*, % and # will be expanded as pathnames (the current and alternative pathnames, respectively), and ! will be replaced with the text of the previous ! or :! command. (Thus, !! will repeat the previous ! command.) If no ! or :! command has yet been executed, an informational message will be displayed stating the problem.

The special meanings of %, # and ! can be overridden by escaping them with a backslash character. This command is affected by the *ex* editor options **autowrite** and **writeany**.

Current line: The first line in *range*.

Current column: The first column of the replaced text.

Move Cursor to End of Line

Synopsis: **[***count***]** $

Move the cursor forward to the end of a line. The *count* argument specifies the number of lines, including the current line, to move forward.

Current line: *line* + *count* – 1.

Current column: *end-of-line*.

Move to Matching Character

Synopsis: %

Move the cursor to the parenthesis or curly brace matching the parenthesis or curly brace at the current position or on the current line forward from the current position. Matching will be determined as follows: for a left parenthesis (respectively curly brace) the editing buffer will be searched forward until either a left or right parenthesis is found. If a left parenthesis is found, a counter is incremented by one and if a right parenthesis is found, a counter is decremented by one, and the search continues (until the end of the editing buffer is found). If a right parenthesis is found when the count is less than or equal to zero, it is the matching parenthesis. For a right parenthesis (respectively curly brace) the editing buffer will be searched backward (to the beginning of the editing buffer), and each right parenthesis increments the counter, and each left parenthesis decrements it. When searching for parentheses, curly braces are not counted, and vice versa. If there is no matching character, the terminal will be alerted and the current position will remain unchanged.

Current line: Move to the line containing the matching character, as described above.

Current column: Move to the column containing the matching character, as described above.

Repeat Substitution

Synopsis: &

Repeat the previous substitution command (equivalent to the *ex* & command) using the current line as the target. Flags specified on the previous substitution command will be ignored by this command.

Current line: Unchanged.

Current column: Unchanged if the previous current column indicator value was *end-of-line*; otherwise, move to the first non-blank character of the current line or the first character if the line is a blank line.

Return to Previous Context at Beginning of Line

Synopsis: ''
Synopsis: ' *letter*

Return to the previous context when followed by a ' character, if the context still exists, placing the cursor at the first non-blank character of the line or the last character if the line is a blank line. If the previous context no longer exists, the ' command will have no effect. When followed by a lower-case letter from the POSIX locale, return to the line marked with that letter (see the m command), at the first non-blank 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 **Return to Previous Context**.)

Current line: Move to the line from the previous context.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Return to Previous Context

Synopsis: ``
Synopsis: ` *letter*

When followed by a ` character, return to the previous context, placing the cursor at the character position marked. (The previous context will be set whenever a non-relative move is made.) When followed by a lower-case letter from the POSIX locale, return 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 **Return to Previous Context at Beginning of Line**.)

Current line: Move to the line from the previous context.

Current column: Set to the position of the character marked from the previous context.

Return to Previous Section

Synopsis: `[[`

Back up to the previous section boundary. This command is affected by the *ex* **sections** option. Note that the brackets in this command are part of the command syntax itself and are not metacharacters.

EX If the *lisp* option is set, a section boundary is also identified by a line with a leading (.

Current line: Move to the previous line that is a section boundary or to the first line of the editing buffer if no more section boundaries exist preceding the current line.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Move to Next Section

Synopsis: `]]`

Move forward to a section boundary. This command is affected by the *ex* **sections** option. Note that the brackets in this command are part of the command syntax itself and are not metacharacters.

EX If the *lisp* option is set, a section boundary is also identified by a line with a leading (.

Current line: Move to the next line that is a section boundary or to the last line of the editing buffer if no more section boundaries exist following the current line.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Move to First Non-blank Position on Current Line

Synopsis: `^`

Move to the first non-blank position on the current line.

Current line: Unchanged.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Move Back to Beginning of Sentence

Synopsis: `[`*count*`] (`

Move backward to the beginning of a sentence. A *count* will move back that many sentences.

EX If the *lisp* option is set, a *lisp* s-expression is considered a sentence for this command.

Current line: Move to the line containing the beginning of the sentence.

Current column: Move to the first non-blank character of the sentence.

Move Forward to Beginning of Sentence

Synopsis: **[***count***]**)

Move forward to the beginning of a sentence. A *count* will move forward that many sentences.

EX If the *lisp* option is set, a *lisp* s-expression is considered a sentence for this command.

Current line: Move to the line containing the beginning of the sentence.

Current column: Move to the first non-blank character of the sentence.

Move Back to Preceding Paragraph

Synopsis: **[***count***]**{

Move back to the beginning of the preceding paragraph. A *count* specifies the number of paragraphs to move backward. This command is affected by the *ex* **paragraph** option.

Current line: Move to the line containing the beginning of the previous paragraph.

Current column: Move to the first non-blank character of the current line.

Move Forward to Next Paragraph

Synopsis: **[***count***]** }

Move forward to the beginning of the next paragraph. A *count* specifies the number of paragraphs to move forward. This command is affected by the *ex* **paragraph** option.

Current line: Move to the line containing the beginning of the next paragraph.

Current column: Move to the first non-blank character of the current line.

Move to Specific Column Position

Synopsis: **[***count***]** |

Place the cursor in the column position identified by *count* (if that position exists on the line). If *count* is omitted, it defaults to 1. If the length of the current line is less than *count*, the cursor will be placed at the end of the current line. If the column position is spanned by a tab or a wide character, the cursor will be placed on the character following the *count*-th column position.

Current line: Unchanged.

Current column: Move to the column position represented by *count*.

Reverse Find Character

Synopsis: **[***count***]** ,

Reverse the last f, F, t or T command, looking the other way in the current line. A *count* will be equivalent to repeating the search that many times.

Current line: Unchanged.

Current column: Move to the first column position of the next character found.

Move to First Non-blank of Previous Line

Synopsis: **[***count***]** **-**

Move to the first non-blank character in the previous line. A *count* specifies how many lines to move back.

Current line: *line – count*

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Repeat

Synopsis: **[***count***]** **.**

Repeat the last command that changed the buffer. A *count* will be passed on to the command being repeated.

Current line: Dependent on the results of the execution of the last command that changed the buffer.

Current column: Dependent on the results of the execution of the last command that changed the buffer.

Find Regular Expression

Synopsis: **/**

Prompt the user to enter a string on the last line on the screen, interpret it as a regular expression (see **Regular Expressions in ex** on page 316), and scan forward for the next occurrence of a matching string. The search will begin when a newline character or an escape character is entered to terminate the pattern; it can be prematurely terminated with an interrupt.

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 can be specified by giving an offset from the matched line (using a closing / followed by a +*n* or –*n*).

Current line: Move to the line in which the first (or next) regular expression match occurred.

Current column: Move to the column of the first character of the matched string.

Move to First Character in Line

Synopsis: **0** (zero)

Move to the first character on the current line. (The zero will not be interpreted as a command when it is preceded by a non-zero digit.)

Current line: Unchanged.

Current column: Column position 1.

Execute an ex Command

Synopsis: :

Execute an *ex* command. The implementation need not support an *ex* command that enters input mode (that is, **append**, **change** or **insert**). The **:** character, as well as the entered command, will be displayed on the bottom line. The command will be executed when the input is terminated by entering a newline character or an escape character. Typing a <control>-V (or <control>-Q) character will escape the following character, allowing its literal value to be entered.

Note: This special property of <control>-V and <control>-Q is only effective in visual or open modes and in *ex* commands called from visual or open mode with the : command.

If the **ex** command causes the screen to be scrolled or causes an escape to the shell, **vi** will display a message indicating that it is waiting for a newline character. When a newline or an escape character is entered, the screen will be refreshed.

When executing an individual *ex* command by entering :, it is not possible to enter a newline character as part of the command because it is considered the end of the command. As explained in **Replacement Strings in ex** on page 316, a <control>-M preceded by a <control>-V (or <control>-Q) escape character in a regular expression substitution is mapped into a newline character. A different approach is to enter *ex* command mode by using the *vi* Q command (and later resuming visual mode with the *ex* vi command). In *ex* command mode, the single-line limitation does not exist. So, for example, the following is valid:

```
Q
s/break here/break\
here/
vi
```

Current line: Dependent on the results of the execution of the *ex* command.

Current column: Dependent on the results of the execution of the *ex* command.

Repeat Find

Synopsis: **[***count***]** **;**

Repeat the last character find using f, F, t or T. A *count* will move the cursor to (or next to) the *count*-th occurrence of the character.

Current line: Unchanged.

Current column: Move to the column containing the character being searched for. If no matching character is found, it will be unchanged.

Shift Left

Synopsis: **[***count***]** **<** *motion*

Shift lines left one *shiftwidth* (see the *ex* **shiftwidth** option). The command must be followed by a motion command to specify lines. A *count* will be passed through to the motion command.

When *motion* is <, it will shift the current line (or *count* lines starting at the current one).

Current line: Unchanged.

Current column: Move to the first non-blank character of the line or the last character if the line is a blank line.

Shift Right

Synopsis: **[***count***]** **>** *motion*

Shift lines right one *shiftwidth* (see the *ex* **shiftwidth** option). The command must be followed by a motion command to specify lines. A *count* will be passed through to the motion command.

When *motion* is >, it will shift the current line (or *count* lines starting at the current one).

Empty lines will not be changed.

Current line: Unchanged.

Current column: Move to the first non-blank character of the line or the last character if the line is a blank line.

Scan Backwards for Regular Expression

Synopsis: ?

Scan backwards, the reverse of /.

Current line: Move to the line in which the next succeeding regular expression match occurred.

Current column: Move to the column of the first character of the matched string.

Execute

Synopsis: @*buffer*

Execute each line of the named buffer as one or more *vi* commands (including *ex* commands, as described in **Execute an ex Command** on page 790). If the buffer name is @, then the last buffer executed is used; if there is no last buffer, an error will occur. The text of a macro may contain an @ character calling another macro; however, recursively calling the same macro produces undefined behaviour. A *vi* error will terminate all currently executing macros. All changes made during a macro call will be treated as a unit and can be undone with a single u command.

Current line: Dependent on the results of the execution of the *vi* commands.

Current column: Dependent on the results of the execution of the *vi* commands.

Reverse Case

Synopsis: **[***count***]** ~

Reverse the case of the *count* characters beginning at the current cursor position. Lower-case alphabetic characters will be changed to upper-case and upper-case characters changed to lower-case. This command will have no effect on non-alphabetic characters.

Current line: Unchanged.

Current column: Move to the next column unless it is on the last character of the line, in which case it will remain unchanged.

Reindent

EX *Synopsis*: **[***count***]=[***motion***]**

If the *lisp* option is set, reindents the specified lines, as though they were typed in with **lisp** and **autoindent** set.

Current line: Unchanged.

Current column: Move to the first non-blank character of the line or the last character if the line is a blank line.

Append

Synopsis: **[***count***]** a

Enter input mode, appending the entered text after the current cursor position. A *count* will cause the inserted text up to the first newline character to be replicated that many times; if a newline character appears in the inserted text, only one occurrence of it and any characters following it will be inserted. For example, if the current line is **abc**, the command:

```
03afoo<newline>bar
```

changes that line to:

```
afoofoofoo
barbc
```

Current line: Unchanged.

Current column: *column* + 1.

Append at End of Line

Synopsis: **[***count***]** A

Append *count* copies of the input text at the end of the current line, equivalent to $**[***count***]** a.

Current line: Unchanged.

Current column: See the a command.

Move Backward to Preceding Word

Synopsis: **[***count***]** b

Move the cursor backward to the beginning of a word by repeating the following algorithm *count* times: If the current position is at the beginning of a word, the current position will move to the first character of the preceding word. Otherwise, the current position will move to the first character of the word at the current position. If no preceding word exists on the current line, the current position will move to the first character of the last word on the first preceding line that contains a word. For this command, an empty or blank line will be considered to contain exactly one word.

Current line: Set to the line containing the word selected.

Current column: Set to the first character of the word selected.

Move Backward to Preceding Bigword

Synopsis: **[***count***]** B

Move the cursor backward to the beginning of a bigword by repeating the following algorithm *count* times: If the current position is at the beginning of a bigword or the character at the current position cannot be part of a bigword, the current position will move to the first character of the preceding bigword. Otherwise, the current position will move to the first character of the bigword at the current position. If no preceding bigword exists on the current line, the current position will move to the first character of the last bigword on the first preceding line that contains a bigword. For this command, an empty or blank line is considered to contain exactly one bigword.

Current line: Set to the line containing the bigword selected.

Current column: Set to the first character of the bigword selected.

Change

Synopsis: **[***buffer***][***count***]** c *motion*

Delete the specified region of text and enter input mode to replace it with the entered text. If more than part of a single line is affected, the deleted text will be saved in the numeric buffers. If only part of the current line is affected, then the last character to be deleted will be marked with a $. A *count* will be passed through to the motion command.

Current line: The current line and column position are dependent on the *motion* command following the c. For example, **cw** changes a word, c[[changes from the beginning of the current section, and so forth.

Current column: See *Current line*, above

Change to End of Line

Synopsis: **[***buffer***][***count***]** C

Change text from the current position to the end of line (*line* + *count* – 1); equivalent to:

[*buffer***][***count***]** c$

Current line: See the c command.

Current column: See the c command.

Delete

Synopsis: **[***buffer***][***count***]** d *motion*

Delete the specified region of text. If more than part of a line is affected, the text will be saved in the numeric buffers as well as any named *buffer*. A *count* will be passed through to the motion command.

Current line: The current line and column position are dependent on the *motion* command following the d. For example, **dw** deletes a word, d$ deletes to the end of the line, and so forth.

Current column: See *Current line*, above.

Delete to End of Line

Synopsis: **[***buffer***]** D

Delete the text from the current position to the end of the current line; equivalent to:

[*buffer***]** d$

Current line: Unchanged.

Current column: Set to maximum of *column* – 1 or 1.

Move to End of Word

Synopsis: **[***count***]** e

Move the cursor forward to the end of a word, by repeating the following algorithm *count* times: If the current position is the end of a word, the current position will move to the last character of the following word. Otherwise, the current position will move to the last character of the word at the current position. If no succeeding word exists on the current line, the current position will move to the last character of the first word on the next following line that contains a word. For this command, an empty or blank line is considered to contain exactly one word.

Current line: Set to the line containing the word selected.

Current column: Set to the last character of the word selected.

Move to End of Bigword

Synopsis: **[***count***]** E

Move the cursor forward to the end of a bigword, by repeating the following algorithm *count* times: If the current position is the end of a bigword or the character at that position cannot be part of a bigword, the current position will move to the last character of the following bigword. Otherwise, the current position will move to the last character of the bigword at the current position. If no succeeding bigword exists on the current line, the current position will move to the last character of the first bigword on the next following line that contains a bigword.

For this command, an empty or blank line is considered to contain exactly one bigword.

Current line: Set to the line containing the bigword selected.

Current column: Set to the last character of the bigword selected.

Find Character in Current Line (Forward)

Synopsis: **[***count***]** f *character*

Scan the rest of the current line for the single character *character* and move the cursor to it if it is found. A *count* will repeat the find that many times.

Current line: Unchanged.

Current column: Set to the column position containing the character that was scanned for. Unchanged if insufficient characters were found to satisfy the *count*.

Find Character in Current Line (Reverse)

Synopsis: **[***count***]** F *character*

Scan backwards in the current line for the single character *character*, moving the cursor to it if it is found. A *count* will be equivalent to repeating the search that many times. If the search is unsuccessful, the terminal will be alerted and the current column position will remain unchanged.

Current line: Unchanged.

Current column: Set to the last character found moving backwards.

Move to Line

Synopsis: **[***count***]** G

Move the cursor to line *count*, or to the last line of the editing buffer if *count* is not specified.

Current line: *count*, if specified; otherwise, the last line.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Move to Top of Screen

Synopsis: **[***count***]** H

Move the cursor to the line (*count* – 1) lines from the top of the current window (that is, the *count*-th line, one-based, currently displayed).

Current line: *count* on the screen, as described above.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Insert Before Cursor

Synopsis: **[***count***]** i

Enter input mode, inserting the entered text before the cursor. (See also the a command.) A *count* will cause the inserted text up to the first newline character to be replicated that many times; if a newline character appears in the inserted text, only one occurrence of it and any characters following it will be inserted.

Current line: Unchanged.

Current column: Unchanged.

Insert at Beginning of Line

Synopsis: **[***count***]** I

Enter input mode at the beginning of a line. This command behaves identically to the **^[***count***]i** command.

Current line: Unchanged.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Join

Synopsis: **[***count***]** J

Join the current line with the next one, supplying appropriate spacing (in the POSIX locale): one space character between words, two space characters after a period, and no space characters at all when the last character of the first line is a blank character or when the first character of the next line is). In the case of lines ending with blank characters, the blanks characters will be retained, no spaces will be added, and the lines will be joined together. A *count* will cause that many lines (minimum 2) to be joined, rather than 2. A count of 1 will be treated as 2.

Current line: Unchanged.

Current column: Move to the character after the last character of the original line.

Move to Bottom of Screen

Synopsis: **[***count***]** L

Move the cursor to the first non-blank character of the last line on the screen. A *count* will move to that line counting from the bottom. When used with an operator, whole lines are affected.

Current line: *last line – count*

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Mark Position

Synopsis: m *letter*

Mark the current position of the cursor with a single POSIX locale lower-case letter *letter*. The exact position is referred to by ‘ *letter*; the line is referred to by ’*letter*.

Current line: Unchanged.

Current column: Unchanged.

Move to Middle of Screen

Synopsis: M

Move the cursor to the middle line on the screen, at the first non-blank position on the line.

Current line: Move to a line approximately in the middle of the display window.

Current column: Move to the first non-blank character of the current line or the last character if the line is a blank line.

Repeat Regular Expression Find (Forward)

Synopsis: n

Repeat the last **/** or **?** scanning command.

Current line: Set to the line containing the character string found by the scan. Unchanged if no match was found.

Current column: Move to the first character of the character string found by the scan. Unchanged if no match was found.

Repeat Regular Expression Find (Reverse)

Synopsis: `N`

Scan for the next match of the last pattern given to **/** or **?**, but in the reverse direction; this is the reverse of **n**.

Current line: Move to the matched line if a match is found or unchanged if no match is found.

Current column: Move to the first column position of the matched string or unchanged if no match is found.

Insert Empty Line Below

Synopsis: `o`

Open a line below the current line and enter input mode.

Current line: The newly-created line, (*line*+1).

Current column: If the editor option **autoindent** is not set, move to the first column; otherwise, as in the description of the *ex* **autoindent** option.

Insert Empty Line Above

Synopsis: `O`

Open a new line above the current line and enter input mode.

Current line: Unchanged.

Current column: If the editor option **autoindent** is not set, move to the first column; otherwise, as described in the description of **autoindent** in *ex*.

Put from Buffer Following

Synopsis: `[`*buffer*`] p`

Insert text from buffer *buffer* following the current column or current line, as described for the P command. If *buffer* is omitted, the unnamed buffer is used.

Current line: (*Current line*+1) if the buffer contains whole lines; otherwise, unchanged.

Current column: First non-blank character of the inserted text if the buffer contained whole lines; otherwise, the last character of the inserted text.

Put from Buffer Before

Synopsis: `[`*buffer*`] P`

Put the last deleted text back before and above the cursor. The text will go back as whole lines above the cursor if it was deleted as whole lines. Otherwise, the text will be inserted just before the cursor.

The command can be preceded by a named buffer specification ("*x*), to retrieve the contents of the buffer.

Current line: Unchanged.

Current column: Move to the last column position of the inserted characters. If the buffer contains whole lines, the current column will be the first non-blank character of the inserted characters.

Enter ex Mode

Synopsis: `Q`

Quit from *vi* and enter *ex* command mode.

Current line: Unchanged.

Current column: Unchanged.

Replace Character

Synopsis: **[***count***]** r *character*

Replace a character on the screen with the character entered. If *count* is specified, the single character entered will replace the current character and each of the next *count* – 1 characters in the line. (For example, 7rx changes the next seven characters to be **xxxxxxx**). Unlike the R command, entering a newline character will replace the character with a newline character; however, when used with *count*, the next *count* characters in the line will be replaced by a single newline character.

Current line: Unchanged unless the replacement character is a newline character, in which case the current line will be incremented by 1.

Current column: Set to the last column position occupied by the replacement character, unless the replacement character is a newline character, in which case the current column position will be 1.

Replace Characters

Synopsis: `R`

Replace characters on the screen with characters entered. If the end of the existing line is encountered, it will be as if insert mode was entered at that point. If a newline character is entered before the end of the existing line, it will be entered into the editing buffer as if it was inserted, and replacement will continue on the newly created line. No other line but the one in which the R command was given will be affected.

Current line: Set to the line at the end of the replaced text.

Current column: Set to the end of the replaced text, if text was replaced. If no text was replaced, set to the column position of the character after which text would have been replaced if that character is on the current line; otherwise, 1.

Substitute Character

Synopsis: **[***buffer***][***count***]** s

Substitute *count* characters in the current line starting with the current column position; equivalent to:

[*buffer***][***count***]** c<space>

Current line: Unchanged.

Current column: Unchanged.

Substitute lines

Synopsis: **[***buffer***][***count***]** S

Change whole lines (same as **cc**). A *count* will change that many lines; *count* lines will be deleted and *vi* will enter input mode.

Current line: Unchanged.

Current column: Set to column position 1 or to the position indicated by the previous line if **autoindent** is set.

Move Cursor to Before Character (Forward)

Synopsis: **[***count***]** t *character*

Move the cursor forward within the current line to the character position just before a subsequent occurrence of *character*. A *count* will place the cursor just before the *count*-th occurrence of the character when searching the line forwards. If the search is unsuccessful, the terminal will be alerted and the cursor position will remain unchanged.

Current line: Unchanged.

Current column: Move to the position before an occurrence of *character* following *column* (non-inclusive).

Move Cursor to After Character (Reverse)

Synopsis: **[***count***]** T *character*

Scan backwards in the current line for the single character *character* and if found, place the cursor just after that character. A *count* will place the cursor just after the *count*-th occurrence of the character when searching the line backwards. If *count*-th occurrences of the character do not exist, the terminal will be alerted and the current column position will remain unchanged.

Current line: Unchanged.

Current column: Set to the column after the scanned character. Unchanged if the character was not found.

Undo

Synopsis: u

Reverse the last change made to the current buffer. If repeated, the command will alternate between these two states, thus is its own inverse. When used after an insert that inserted text on more than one line, the lines will be saved in the numeric named buffers.

Current line: Move to the position of the first line changed, if the reversal affects only one line or represents an addition or change; otherwise, move to the line preceding the deleted text.

Current column: Move to the first non-blank character on the updated current line.

Undo Current Line

Synopsis: U

Restore the current line to its state before the cursor was last moved to it.

Current line: Unchanged.

Current column: Set to column position 1 or to the position indicated by the previous line if **autoindent** is set.

Move to Beginning of Word

Synopsis: **[***count***]** w

Move the cursor forward to the beginning of a word by repeating the following algorithm *count* times: If the current position is at the beginning of a word, the current position will move to the first character of the next word. If no subsequent word exists on the current line, the current position will move to the first character of the first word on the first following line that contains a word.

For this command, an empty or blank line is considered to contain exactly one word.

Current line: Set to the line containing the word selected.

Current column: Set to the first character of the word selected.

Move to Beginning of Bigword

Synopsis: **[***count***]** W

Move the cursor forward to the beginning of a bigword by repeating the following algorithm *count* times: If the current position is within a bigword or the character at that position cannot be part of a bigword, the current position will move to the first character of the next bigword. If no subsequent bigword exists on the current line, the current position will move to the first character of the first bigword on the first following line that contains a bigword. For this command, an empty or blank line is considered to contain exactly one bigword.

Current line: Set to the line containing the bigword selected.

Current column: Set to the first character of the bigword selected.

Delete Character at Cursor

Synopsis: **[***buffer***][***count***]** x

Delete *count* characters in the current line starting with the current column position; equivalent to:

[*buffer***][***count***]** dl

If the number of characters to be deleted is greater than or equal to the number of characters to the end of the line, delete all of the characters from the current position to the end of the line and move the cursor to the new last character on the line.

Current line: Unchanged.

Current column: Set to the greater of (*current column* – the width of the *count* deleted characters) or 1.

Delete Character Before Cursor

Synopsis: **[***buffer***][***count***]** X

Delete the character before the cursor; equivalent to:

[*buffer***][***count***]** dh

A *count* will repeat the effect, but only characters on the current line will be deleted.

Current line: Unchanged.

Current column: Set to the greater of (*current column* – the width of the *count* deleted characters) or 1.

Yank

Synopsis: **[***buffer***][***count***]** y *motion*

Copy (yank) the area of text specified by *count* and *motion* into the unnamed buffer, as well as into any named buffer specified by *buffer*.

Current line: Unchanged.

Current column: Unchanged.

Yank Current Line

Synopsis: **[***buffer***][***count***]** Y

Copy (yank) *count* lines starting with the current line into the unnamed buffer, as well as into any named buffer specified by *buffer*; equivalent to a:

[*buffer***][***count***]** yy

command.

Current line: Unchanged.

Current column: Unchanged.

Redraw Window

Synopsis: **[***count1***]** z **[***count2***]** *character*

Redraw the screen with a window *count2* lines long containing line *count1* placed as specified by *character*: a newline character specifies the top of the screen, . specifies the center of the screen, and – specifies the bottom of the screen. If *count1* is not specified, it defaults to the current line; if *count2* is not specified, it defaults to *window*.

Current line: *count1*, if specified; otherwise, unchanged.

Current column: Move to the first non-blank character of the current line or the first character if the line is a blank line.

Exit

Synopsis: ZZ

Exits the editor, writing out the buffer if it was changed since the last write to any file. This command behaves identically to the *ex* command **xit**.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The uses described for <control>-V can also be accomplished with <control>-Q, which is useful on terminals that use <control>-V for the down-arrow function. However, most historical implementations use <control>-Q for the **termios** START character, so the editor will generally not receive the <control>-Q unless *stty* –**ixon** mode is set. Any of the command characters described in this document can be made ineffective by their selection as **termios** control characters, using the *stty* utility or other methods described in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

ex.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

wait – await process completion

SYNOPSIS

`wait [`*`pid`*`...]`

DESCRIPTION

When an asynchronous list (see Section 2.9.3 on page 50) is started by the shell, the process ID of the last command in each element of the asynchronous list becomes known in the current shell execution environment; see Section 2.12 on page 63.

If the *wait* utility is invoked with no operands, it will wait until all process IDs known to the invoking shell have terminated and exit with a zero exit status.

If one or more *pid* operands are specified that represent known process IDs, the *wait* utility will wait until all of them have terminated. If one or more *pid* operands are specified that represent unknown process IDs, *wait* will treat them as if they were known process IDs that exited with exit status 127. The exit status returned by the *wait* utility will be the exit status of the process requested by the last *pid* operand.

The known process IDs are applicable only for invocations of *wait* in the current shell execution environment.

OPTIONS

None.

OPERANDS

The following operand is supported:

pid One of the following:

1. The unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.
2. A job control job ID (see the **XBD** specification, **Chapter 2**, **Glossary**) that identifies a background process group to be waited for. The job control job ID notation is applicable only for invocations of *wait* in the current shell execution environment; see Section 2.12 on page 63. The exit status of *wait* is determined by the last command in the pipeline.

   Note: The job control job ID type of *pid* is available on systems supporting both the job control option and the User Portability Utilities Option, which applies to all X/Open systems.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *wait*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
Not used.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
If one or more operands were specified, all of them have terminated or were not known by the invoking shell, and the status of the last operand specified is known, then the exit status of *wait* will be the exit status information of the command indicated by the last operand specified. If the process terminated abnormally due to the receipt of a signal, the exit status will be greater than 128 and will be distinct from the exit status generated by other signals, but the exact value is unspecified. (See the *kill* **–l** option.) Otherwise, the *wait* utility will exit with one of the following values:

0 The *wait* utility was invoked with no operands and all process IDs known by the invoking shell have terminated.
1–126 The *wait* utility detected an error.
127 The command identified by the last *pid* operand specified is unknown.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
On most implementations, *wait* is a shell built-in. If it is called in a subshell or separate utility execution environment, such as one of the following:

```
(wait)
nohup wait ...
find . -exec wait ... \;
```

it will return immediately because there will be no known process IDs to wait for in those environments.

Historical implementations of interactive shells have discarded the exit status of terminated background processes before each shell prompt. Therefore, the status of background processes

was usually lost unless it terminated while *wait* was waiting for it. This could be a serious problem when a job that was expected to run for a long time actually terminated quickly with a syntax or initialisation error because the exit status returned was usually zero if the requested process ID was not found. This document requires the implementation to keep the status of terminated jobs available until the status is requested, so that scripts like:

```
j1&
p1=$!
j2&
wait $p1
echo Job 1 exited with status $?
wait $!
echo Job 2 exited with status $?
```

will work without losing status on any of the jobs. The shell is allowed to discard the status of any process that it determines the application cannot get the process ID from the shell. It is also required to remember only {CHILD_MAX} number of processes in this way. Since the only way to get the process ID from the shell is by using the ! shell parameter, the shell is allowed to discard the status of an asynchronous list if $! was not referenced before another asynchronous list was started. (This means that the shell only has to keep the status of the last asynchronous list started if the application did not reference $!. If the implementation of the shell is smart enough to determine that a reference to $! was not saved anywhere that the application can retrieve it later, it can use this information to trim the list of saved information. Note also that a successful call to *wait* with no operands discards the exit status of all asynchronous lists.)

If the exit status of *wait* is greater than 128, there is no way for the application to know if the waited-for process exited with that value or was killed by a signal. Since most utilities exit with small values, there is seldom any ambiguity. Even in the ambiguous cases, most applications just need to know that the asynchronous job failed; it does not matter whether it detected an error and failed or was killed and did not complete its job normally.

EXAMPLES

Although the exact value used when a process is terminated by a signal is unspecified, if it is known that a signal terminated a process, a script can still reliably figure out which signal using *kill* as shown by the following script:

```
sleep 1000&
pid=$!
kill -kill $pid
wait $pid
echo $pid was terminated by a SIG$(kill -l $?) signal.
```

If the following sequence of commands is run in less than 31 seconds:

```
sleep 257 | sleep 31 &
jobs -l %%
```

either of the following commands will return the exit status of the second *sleep* in the pipeline:

```
wait <pid of sleep 31>
wait %%
```

FUTURE DIRECTIONS

None.

SEE ALSO

sh, the **XSH** specification description of *waitpid*( ).

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

wall – write to all users (**WITHDRAWN**)

SYNOPSIS

EX `wall`

APPLICATION USAGE

This utility has been withdrawn because it cannot be used portably by an application. It is usually usable only by applications with appropriate privileges.

SEE ALSO

mesg, *write*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

This page has been marked as withdrawn.

NAME

wc – word, line and byte or character count

SYNOPSIS

```
wc [ -c| -m[-lw][file...]
```

DESCRIPTION

The *wc* utility reads one or more input files and, by default, writes the number of newline characters, words and bytes contained in each input file to the standard output.

The utility also writes a total count for all named files, if more than one input file is specified.

The *wc* utility considers a *word* to be a non-zero-length string of characters delimited by white space.

OPTIONS

The *wc* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–c Write to the standard output the number of bytes in each input file.

–l Write to the standard output the number of newline characters in each input file.

–m Write to the standard output the number of characters in each input file.

–w Write to the standard output the number of words in each input file.

When any option is specified, *wc* will report only the information requested by the specified options.

OPERANDS

The following operand is supported:

file A pathname of an input file. If no *file* operands are specified, the standard input will be used.

STDIN

The standard input will be used only if no *file* operands are specified. See **INPUT FILES**.

INPUT FILES

The input files may be of any type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *wc*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and which characters are defined as white space characters.

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS
Default.

STDOUT
By default, the standard output contains an entry for each input file of the form:

```
"%d %d %d %s\n", <newlines>, <words>, <bytes>, <file>
```

If the **–m** option is specified, the number of characters replace the <*bytes*> field in this format.

If any options are specified and the **–l** option is not specified, the number of newline characters will not be written.

If any options are specified and the **–w** option is not specified, the number of words will not be written.

If any options are specified and neither **–c** nor **–m** is specified, the number of bytes or characters will not be written.

If no input *file* operands are specified, no name will be written and no blank characters preceding the pathname will be written.

If more than one input *file* operand is specified, an additional line will be written, of the same format as the other lines, except that the word **total** (in the POSIX locale) will be written instead of a pathname and the total of each column will be written as appropriate. Such an additional line, if any, will be written at the end of the output.

STDERR
Used only for diagnostic messages.

OUTPUT FILES
None.

EXTENDED DESCRIPTION
None.

EXIT STATUS
The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS
Default.

APPLICATION USAGE
The **–m** option is not a switch, but an option at the same level as **–c**. Thus, to produce the full default output with character counts instead of bytes, the command required is:

```
wc -mlw
```

EXAMPLES
None.

FUTURE DIRECTIONS

None.

SEE ALSO

cksum.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

what – identify SCCS files (**DEVELOPMENT**)

SYNOPSIS

EX `what [-s] file...`

DESCRIPTION

The *what* utility searches the given files for all occurrences of the pattern that *get* (see *get*) substitutes for **%Z%** (@(#)) and writes to standard output what follows until the first ", >, newline, \ or NUL character.

OPTIONS

The *what* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**. The following option is supported:

–s Quit after finding the first occurrence of the pattern in each file.

OPERANDS

The following operands are supported:

file A pathname of a file to search.

STDIN

Not used.

INPUT FILES

The input files are of any file type.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *what*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The standard output consists of the following for each *file* operand:

`"%s:\n\t%s\n", <pathname>, <identification string>`

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Any matches were found.
1 Otherwise.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *what* utility is intended to be used in conjunction with the SCCS command *get*, which automatically inserts identifying information, but it can also be used where the information is inserted by any other means.

When a **what** string is included in a library routine in a shared library, it might not be found in an **a.out** file using that library routine.

EXAMPLES

If the C-language program in file **f.c** contains:

```
char ident[] = "@(#)identification information";
```

and **f.c** is compiled to yield **f.o** and **a.out**, then the command:

```
what f.c f.o a.out
```

will write:

```
f.c:
        identification information
        ...
f.o:
        identification information
        ...
a.out:
        identification information
        ...
```

FUTURE DIRECTIONS

None.

SEE ALSO

get.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Format reorganised.

Utility Syntax Guidelines support mandated.

Internationalised environment variable support mandated.

NAME

who – display who is on the system

SYNOPSIS

EX `who [-mu]-s[-bHlprt][file]`

EX `who [-mTu][-abdHlprt][file]`

EX `who -q [file]`

EX `who am i`

EX `who am I`

DESCRIPTION

The *who* utility lists various pieces of information about accessible users. The domain of accessibility is implementation-dependent.

EX Based on the options given, *who* also 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.

OPTIONS

The *who* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported. The metavariables, such as <*line*>, refer to fields described in **STDOUT**.

EX **–a** Process the implementation-dependent database or named file with the **–b**, **–d**, **–l**, **–p**, **–r**, **–t**, **–T** and **–u** options turned on.

EX **–b** Write the time and date of the last reboot.

EX **–d** Write a list of all processes that have expired and not been respawned by the *init* system process. 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.

EX **–H** Write column headings above the regular output.

EX **–l** (The letter ell.) List 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.

–m Output only information about the current terminal.

EX **–p** List any other process that is currently active and has been previously spawned by *init*.

EX **–q** (Quick.) List only the names and the number of users currently logged on. When this option is used, all other options are ignored.

EX **–r** Write the current *run-level* of the *init* process.

EX **–s** List only the <*name*>, <*line*> and <*time*> fields. This is the default case.

EX **–t** Indicate the last change to the system clock.

–T Show the state of each terminal, as described in **STDOUT**.

EX **–u** This option lists only those users who are currently logged in. Output the user's ''idle time'' in addition to any other information. The idle time is the time since any activity EX occurred on the user's terminal. The method of determining this is unspecified. 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.

OPERANDS

EX The following operands are supported:

am i
am I In the POSIX locale, limit the output to describing the invoking user, equivalent to the **–m** option. The **am** and i or I must be separate arguments.

file Specify a pathname of a file to substitute for the implementation-dependent database of logged-on users that *who* uses by default.

STDIN

Not used.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *who*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES
Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

LC_TIME
Determine the locale used for the format and contents of the date and time strings.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

EX OF The *who* utility writes its default information to the standard output in the following general format:

```
<name>[<state>]<line><time>[<activity>][<pid>][<comment>][<exit>]
```

The following format is used for the –**T** option:

```
"%s %c %s %s\n", <name>, <terminal state>, <terminal name>,
<time of login>
```

where <*terminal state*> is one of the following characters:

+ The terminal allows write access to other users.

– The terminal denies write access to other users.

? The terminal write-access state cannot be determined.

In the POSIX locale, the <*time of login*> is equivalent in format to the output of:

```
date +"%b %e %H:%M"
```

If the –**u** option is used with –**T**, the idle time is added to the end of the previous format in an unspecified format.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The name *init* used for the system process is the most commonly used on historical systems, but it may vary.

The ‘‘domain of accessibility’’ referred to is a broad concept that permits interpretation either on a very secure basis or even to allow a network-wide implementation like the historical `rwho`.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

write – write to another user

SYNOPSIS

`write` *user_name* **[***terminal***]**

DESCRIPTION

The *write* utility reads lines from the user's standard input and writes them to the terminal of another user. When first invoked, it writes the message:

Message from *sender-login-id* (*sending-terminal*) **[***date***]...**

to *user_name*. When it has successfully completed the connection, the sender's terminal will be alerted twice to indicate that what the sender is typing is being written to the recipient's terminal.

If the recipient wants to reply, this can be accomplished by typing:

`write` *sender-login-id* **[***sending-terminal***]**

upon receipt of the initial message. Whenever a line of input as delimited by a NL, EOF or EOL special character (see the **XBD** specification, **Chapter 9**, **General Terminal Interface**) is accumulated while in canonical input mode, the accumulated data will be written on the other user's terminal. Characters are processed as follows:

- Typing the alert character will write the alert character to the recipient's terminal.
- Typing the erase and kill characters will affect the sender's terminal in the manner described by the **termios** interface in the **XBD** specification, **Chapter 9**, **General Terminal Interface**.
- Typing the interrupt or end-of-file characters will cause *write* to write an appropriate message (EOT\n in the POSIX locale) to the recipient's terminal and exit.
- Typing characters from LC_CTYPE classifications **print** or **space** will cause those characters to be sent to the recipient's terminal.
- When and only when the *stty iexten* local mode is enabled, the existence and processing of additional special control characters and multi-byte or single-byte functions is implementation-dependent.
- Typing other non-printable characters will cause implementation-dependent sequences of printable characters to be written to the recipient's terminal.

To write to a user who is logged in more than once, the *terminal* argument can be used to indicate which terminal to write to; otherwise, the recipient's terminal is selected in an implementation-dependent manner and an informational message will be written to the sender's standard output, indicating which terminal was chosen.

Permission to be a recipient of a *write* message can be denied or granted by use of the *mesg* utility. However, a user's privilege may further constrain the domain of accessibility of other users' terminals. The *write* utility will fail when the user lacks the appropriate privileges to perform the requested action.

OPTIONS

None.

OPERANDS

The following operands are supported:

user_name
: Login name of the person to whom the message will be written. This operand must be of the form returned by the *who* utility.

terminal
: Terminal identification in the same format provided by the *who* utility.

STDIN

Lines to be copied to the recipient's terminal will be read from standard input.

INPUT FILES

None.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *write*:

LANG
: Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
: If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE
: Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files). If the recipient's locale does not use an LC_CTYPE equivalent to the sender's, the results are undefined.

LC_MESSAGES
: Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error and informative messages written to standard output.

EX *NLSPATH*
: Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

If an interrupt signal is received, *write* will write an appropriate message on the recipient's terminal and exit with a status of zero. It will take the standard action for all other signals.

STDOUT

An informational message will be written to standard output if a recipient is logged in more than once.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

The recipient's terminal will be used for output.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 The addressed user is not logged on or the addressed user denies permission.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

The *talk* utility is considered by some users to be a more usable utility on full-screen terminals.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

mesg, *talk*, *who*, the **XBD** specification, **Chapter 9**, **General Terminal Interface**.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

xargs – construct argument lists and invoke utility

SYNOPSIS

EX OB
```
xargs [-t][-p][-e[eofstr]][-E eofstr][-I replstr][-i[replstr]]
[-L number][-l[number]][-n number [-x]][-s size][utility [argument...]]
```

DESCRIPTION

The *xargs* utility constructs a command line consisting of the *utility* and *argument* operands specified followed by as many arguments read in sequence from standard input as will fit in length and number constraints specified by the options. The *xargs* utility then invokes the constructed command line and waits for its completion. This sequence is repeated until an end-of-file condition is detected on standard input or an invocation of a constructed command line returns an exit status of 255.

Arguments in the standard input must be separated by unquoted blank characters, or unescaped blank characters or newline characters. A string of zero or more non-double-quote (") and non-newline characters can be quoted by enclosing them in double-quotes. A string of zero or more non-apostrophe (') and non-newline characters can be quoted by enclosing them in apostrophes. Any unquoted character can be escaped by preceding it with a backslash. The *utility* will be executed one or more times until the end-of-file is reached. The results are unspecified if the utility named by *utility* attempts to read from its standard input.

The generated command line length will be the sum of the size in bytes of the utility name and each argument treated as strings, including a null byte terminator for each of these strings. The *xargs* utility will limit the command line length such that when the command line is invoked, the combined argument and environment lists (see the *exec* family of functions in the **XSH** specification) will not exceed {ARG_MAX}–2048 bytes. Within this constraint, if neither the **–n** nor the **–s** option is specified, the default command line length will be at least {LINE_MAX}.

OPTIONS

OB The *xargs* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**, except that the **–e**, **–i** and **–l** take optional option-arguments that cannot be separate arguments.

The following options are supported:

EX OB **–e[***eofstr***]**
: Use *eofstr* as the logical end-of-file string. Underscore (_) is assumed for the logical EOF string if neither **–e** nor **–E** is used. When the **–eofstr** option-argument is omitted, the logical EOF string capability is disabled and underscores are taken literally. The *xargs* utility reads standard input until either end-of-file or the logical EOF string is encountered.

EX **–E** *eofstr*
: Specify a logical end-of-file string to replace the default underscore. The *xargs* utility reads standard input until either end-of-file or the logical EOF string is encountered.

EX **–I** *replstr*
: Insert mode: *utility* will be executed for each line from standard input, taking the entire line as a single argument, inserting it in *argument*s for each occurrence of *replstr*. A maximum of five arguments in *argument*s can each contain one or more instances of *replstr*. Any blank characters at the beginning of each line are ignored. Constructed arguments cannot grow larger than 255 bytes. Option **–x** is forced on. The **–I** and **–i** options are mutually exclusive; the last one specified takes effect.

OB **–i[***replstr***]**
This option is equivalent to **–I** *replstr*. The string **{}** is assumed for *replstr* if the option-argument is omitted.

EX **–L** *number*
The *utility* will be executed for each non-empty *number* lines of arguments from standard input. The last invocation of *utility* will be with fewer lines of arguments if fewer than *number* remain. A line is considered to end with the first newline character unless the last character of the line is a blank character; a trailing blank character signals continuation to the next non-empty line, inclusive. The **–L**, **–l** and **–n** options are mutually exclusive; the last one specified takes effect.

OB **–l[***number***]**
(The letter ell.) This option is equivalent to **–L** *number*. If *number* is omitted, 1 is assumed. Option **–x** is forced on.

–n *number*
Invoke *utility* using as many standard input arguments as possible, up to *number* (a positive decimal integer) arguments maximum. Fewer arguments will be used if:

- The command line length accumulated exceeds the size specified by the **–s** option (or {LINE_MAX} if there is no **–s** option), or
- The last iteration has fewer than *number*, but not zero, operands remaining.

EX **–p** Prompt mode: the user is asked whether to execute *utility* at each invocation. Trace mode (**–t**) is turned on to write the command instance to be executed, followed by a prompt to standard error. An affirmative response read from **/dev/tty** will execute the command; otherwise, that particular invocation of *utility* is skipped.

–s *size* Invoke *utility* using as many standard input arguments as possible yielding a command line length less than *size* (a positive decimal integer) bytes. Fewer arguments will be used if:

- The total number of arguments exceeds that specified by the **–n** option, or

EX

- The total number of lines exceeds that specified by the **–L** option, or
- End of file is encountered on standard input before *size* bytes are accumulated.

Values of *size* up to at least {LINE_MAX} bytes are supported, provided that the constraints specified in **DESCRIPTION** are met. It is not considered an error if a value larger than that supported by the implementation or exceeding the constraints specified in **DESCRIPTION** is given; *xargs* will use the largest value it supports within the constraints.

–t Enable trace mode. Each generated command line will be written to standard error just prior to invocation.

EX **–x** Terminate if a command line containing *number* arguments (see the **–n** option above) or *number* lines (see the **–L** option above) will not fit in the implied or specified size (see the **–s** option above).

OPERANDS

The following operands are supported:

utility The name of the utility to be invoked, found by search path using the *PATH* environment variable, described in the **XBD** specification, **Chapter 6**, **Environment Variables**. If *utility* is omitted, the default is the *echo* utility. If the *utility* operand names any of the special built-in utilities in Section 2.14 on page 67, the results are undefined.

argument
An initial option or operand for the invocation of *utility*.

STDIN

The standard input must be a text file. The results are unspecified if an end-of-file condition is detected immediately following an escaped newline character.

INPUT FILES

EX The file **/dev/tty** is used to read responses required by the **–p** option.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *xargs*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL
If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_COLLATE
Determine the locale for the behaviour of ranges, equivalence classes and multi-character collating elements used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_CTYPE
Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files) and the behaviour of character classes used in the extended regular expression defined for the **yesexpr** locale keyword in the LC_MESSAGES category.

LC_MESSAGES
Determine the locale for the processing of affirmative responses and that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*
Determine the location of message catalogues for the processing of *LC_MESSAGES*.

PATH Determine the location of *utility*, as described in the **XBD** specification, **Chapter 6**, **Environment Variables**.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

EX Used for diagnostic messages and the **–t** and **–p** options. If the **–t** option is specified, the *utility* and its constructed argument list will be written to standard error, as it will be invoked, prior to
EX invocation. If **–p** is specified, a prompt of the following format will be written (in the POSIX locale):

```
?...
```

at the end of the line of the output from **–t**.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 All invocations of *utility* returned exit status zero.

1-125 A command line meeting the specified requirements could not be assembled, one or more of the invocations of *utility* returned a non-zero exit status, or some other error occurred.

126 The utility specified by *utility* was found but could not be invoked.

127 The utility specified by *utility* could not be found.

CONSEQUENCES OF ERRORS

If a command line meeting the specified requirements cannot be assembled, the utility cannot be invoked, an invocation of the utility is terminated by a signal, or an invocation of the utility exits with exit status 255, the *xargs* utility will write a diagnostic message and exit without processing any remaining input.

APPLICATION USAGE

The 255 exit status (described as –1 in Issue 3) allows a utility being used by *xargs* to tell *xargs* to terminate if it knows no further invocations using the current data stream will succeed. Thus, *utility* should explicitly *exit* with an appropriate value to avoid accidentally returning with 255.

Note that input is parsed as lines; blank characters separate arguments. If *xargs* is used to bundle output of commands like *find dir* **–print** or *ls* into commands to be executed, unexpected results are likely if any filenames contain any blank characters or newline characters. This can be fixed by using *find* to call a script that converts each file found into a quoted string that is then piped to *xargs*. Note that the quoting rules used by *xargs* are not the same as in the shell. They were not made consistent here because existing applications depend on the current rules and the shell syntax is not fully compatible with it. An easy rule that can be used to transform any string into a quoted form that *xargs* will interpret correctly is to precede each character in the string with a backslash.

On implementations with a large value for {ARG_MAX}, *xargs* may produce command lines longer than {LINE_MAX}. For invocation of utilities, this is not a problem. If *xargs* is being used to create a text file, users should explicitly set the maximum command line length with the **–s** option.

The *command*, *env*, *nice*, *nohup*, *time* and *xargs* utilities have been specified to use exit code 127 if an error occurs so that applications can distinguish ‘‘failure to find a utility’’ from ‘‘invoked utility exited with an error indication.’’ The value 127 was chosen because it is not commonly used for other meanings; most utilities use small values for ‘‘normal error conditions’’ and the values above 128 can be confused with termination due to receipt of a signal. The value 126 was chosen in a similar manner to indicate that the utility could be found, but not invoked. Some

scripts produce meaningful error messages differentiating the 126 and 127 cases. The distinction between exit codes 126 and 127 is based on KornShell practice that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 126 when any attempt to *exec* the utility fails for any other reason.

EXAMPLES

EX 1. 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/{}
```

2. The following command will combine the output of the parenthesised commands onto one line, which is then written to the end of file **log**:

```
(logname; date; printf "%s\n" "$0 $*") | xargs >>log
```

3. The following command will invoke *diff* with successive pairs of arguments originally typed as command line arguments (assuming there are no embedded blank characters in the elements of the original argument list):

```
printf "%s\n" "$*" | xargs -n 2 -x diff
```

EX 4. The user is asked which files in the current directory are to be archived. The files are archived into **arch**; a, one at a time, or b, many at a time.

```
a.  ls | xargs -p -L 1 ar -r arch
```

```
b.  ls | xargs -p -L 1 | xargs ar -r arch
```

5. The following will execute with successive pairs of arguments originally typed as command line arguments:

```
echo $* | xargs -n 2 diff
```

FUTURE DIRECTIONS

A version supporting the Utility Syntax Guidelines may be introduced.

SEE ALSO

echo.

CHANGE HISTORY

First released in Issue 2.

Issue 3

Functionally equivalent to the entry in Issue 2.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

yacc – yet another compiler compiler (**DEVELOPMENT**)

SYNOPSIS

```
yacc [-dltv][-b file_prefix][-p sym_prefix] grammar
```

DESCRIPTION

The *yacc* utility reads a description of a context-free grammar in *file* and writes C source code, conforming to the ISO C standard, to a code file, and optionally header information into a header file, in the current directory. The C code defines a function and related routines and macros for an automaton that executes a parsing algorithm meeting the requirements in **Algorithms** on page 838.

The form and meaning of the grammar are described in **EXTENDED DESCRIPTION**.

The C source code and header file are produced in a form suitable as input for the C compiler (see *c89*).

OPTIONS

The *yacc* utility supports the **XBD** specification, **Section 10.2**, **Utility Syntax Guidelines**.

The following options are supported:

–b *file_prefix*
: Use *file_prefix* instead of y as the prefix for all output filenames. The code file **y.tab.c**, the header file **y.tab.h** (created when **–d** is specified), and the description file **y.output** (created when **–v** is specified), will be changed to *file_prefix***.tab.c**, *file_prefix***.tab.h**, and *file_prefix***.output**, respectively.

–d
: Write the header file; by default only the code file is written. The **#define** statements that associate the token codes assigned by *yacc* with the user-declared token names. This allows source files other than **y.tab.c** to access the token codes.

–l
: Produce a code file that does not contain any **#line** constructs. If this option is not present, it is unspecified whether the code file or header file contains **#line** directives. This should only be used after the grammar and the associated actions are fully debugged.

–p *sym_prefix*
: Use *sym_prefix* instead of yy as the prefix for all external names produced by *yacc*. The names affected include the functions *yyparse*( ), *yylex*( ) and *yyerror*( ), and the variables *yylval*, *yychar* and *yydebug*. (In the remainder of this section, the six symbols cited are referenced using their default names only as a notational convenience.) Local names may also be affected by the **–p** option; however, the **–p** option does not affect **#define** symbols generated by *yacc*.

–t
: Modify conditional compilation directives to permit compilation of debugging code in the code file. Run-time debugging statements will be always contained in the code file, but by default conditional compilation directives prevent their compilation.

–v
: Write a file containing a description of the parser and a report of conflicts generated by ambiguities in the grammar.

OPERANDS

The following operand is required:

grammar

A pathname of a file containing instructions, hereafter called *grammar*, for which a parser is to be created. The format for the grammar is described in **EXTENDED DESCRIPTION**.

STDIN

Not used.

INPUT FILES

The file *grammar* must be a text file formatted as specified in **EXTENDED DESCRIPTION**.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *yacc*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments and input files).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

EX *NLSPATH*

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

The *LANG* and *LC_** variables affect the execution of the *yacc* utility as stated. The *main*() function defined in **Yacc Library** on page 837 calls:

```
setlocale(LC_ALL, "")
```

and thus, the program generated by *yacc* will also be affected by the contents of these variables at runtime.

ASYNCHRONOUS EVENTS

Default.

STDOUT

Not used.

STDERR

If shift/reduce or reduce/reduce conflicts are detected in *grammar*, *yacc* writes a report of those conflicts to the standard error in an unspecified format.

Standard error is also used for diagnostic messages.

OUTPUT FILES

The code file, the header file and the description file are text files. All are described in the following sections.

Code file

This file will contain the C source code for the *yyparse*( ) routine. It will contain code for the various semantic actions with macro substitution performed on them as described in **EXTENDED DESCRIPTION**. It will also contain a copy of the **#define** statements in the header file. If a **%union** declaration is used, the declaration for YYSTYPE also will be included in this file.

The contents of the Program Section (see **Programs Section** on page 832) of the input file will then be included.

Header file

The header file will contain **#define** statements that associate the token numbers with the token names. This allows source files other than the code file to access the token codes. If a **%union** declaration is used, the declaration for YYSTYPE and an extern YYSTYPE yylval declaration also will be included in this file.

Description file

The description file will be a text file containing a description of the state machine corresponding to the parser, using an unspecified format. Limits for internal tables (see **Limits** in **EXTENDED DESCRIPTION**) will also be reported, in an implementation-dependent manner. (Some implementations may use dynamic allocation techniques and have no specific limit values to report.)

EXTENDED DESCRIPTION

The *yacc* command accepts a language that is used to define a grammar for a target language to be parsed by the tables and code generated by *yacc*. The language accepted by *yacc* as a grammar for the target language is described below using the *yacc* input language itself.

The input *grammar* includes rules describing the input structure of the target language and code to be invoked when these rules are recognised to provide the associated semantic action. The code to be executed will appear as bodies of text that are intended to be C-language code. The C-language inclusions are presumed to form a correct function when processed by *yacc* into its output files. The code included in this way will be executed during the recognition of the target language.

Given a grammar, the *yacc* utility generates the files described in **OUTPUT FILES**. The code file can be compiled and linked using *cc* or *c89*. If the declaration and programs sections of the grammar file did not include definitions of *main*( ), *yylex*( ) and *yyerror*( ), the compiled output requires linking with externally supplied version of those functions. Default versions of *main*( ) and *yyerror*( ) are supplied in the *yacc* library and can be linked in by using the **–l y** operand to *cc* or *c89*. The *yacc* library interfaces need not support interfaces with other than the default **yy** symbol prefix. The application provides the lexical analyser function, *yylex*( ); the *lex* utility is specifically designed to generate such a routine.

Input Language

Every specification file must consist of three sections in order: *declarations*, *grammar rules* and *programs*, separated by double percent signs (**%%**). The declarations and programs sections can be empty. If the latter is empty, the preceding **%%** mark separating it from the rules section can be omitted.

The input is free form text following the structure of the grammar defined below.

Lexical Structure of the Grammar

The characters blank, newline and form-feed are ignored, except that they must not appear in names or multi-character reserved symbols. Comments must be enclosed in /*...*/, and can appear wherever a name is valid.

Names are of arbitrary length, made up of letters, periods (.), underscores (_) and non-initial digits. Upper- and lower-case letters are distinct. Portable applications must not use names beginning in yy or YY since the *yacc* parser uses such names. Many of the names appear in the final output of *yacc*, and thus they should be chosen to conform with any additional rules created by the C compiler to be used. In particular they will appear in **#define** statements.

A literal consists of a single character enclosed in single-quotes (’). All of the escape sequences supported for character constants by the ISO C standard are supported by *yacc*.

The relationship with the lexical analyser is discussed in detail below.

The NUL character must not be used in grammar rules or literals.

Declarations Section

The declarations section is used to define the symbols used to define the target language and their relationship with each other. In particular, much of the additional information required to resolve ambiguities in the context-free grammar for the target language is provided here.

Usually *yacc* assigns the relationship between the symbolic names it generates and their underlying numeric value. The declarations section makes it possible to control the assignment of these values.

It is also possible to keep semantic information associated with the tokens currently on the parse stack in a user-defined C-language **union**, if the members of the union are associated with the various names in the grammar. The declarations section provides for this as well.

The first group of declarators below all take a list of names as arguments. That list can optionally be preceded by the name of a C union member (called a *tag* below) appearing within < and >. (As an exception to the typographical conventions of the rest of this document, in this case <*tag*> does not represent a metavariable, but the literal angle bracket characters surrounding a symbol.) The use of *tag* specifies that the tokens named on this line are to be of the same C type as the union member referenced by *tag*. This is discussed in more detail below.

For lists used to define tokens, the first appearance of a given token can be followed by a positive integer (as a string of decimal digits). If this is done, the underlying value assigned to it for lexical purposes will be taken to be that number.

`%token` **[**<*tag*>**]** *name* **[**`number`**]** **[**`name` **[**`number`**]]...**

Declares *name*s to be a token. If *tag* is present, the C type for all tokens on this line will be declared to be the type referenced by *tag*. If a positive integer, *number*, follows a *name*, that value will be assigned to the token.

`%left` **[**<*tag*>**]** *name* **[**`number`**]** **[**`name` **[**`number`**]]...**
`%right` **[**<*tag*>**]** *name* **[**`number`**]** **[**`name` **[**`number`**]]...**

Declares *name* to be a token, and assigns precedence to it. One or more lines, each beginning with one of these symbols, can appear in this section. 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** similarly denotes right associative operators. If *tag* is present, it declares a C type for *name*s as described for **%token**.

`%nonassoc` **[**<*tag*>**]** *name* **[***number***]** **[***name* **[***number***]]**`...`
: Declares *name* to be a token, and indicates that this cannot be used associatively. If the parser encounters associative use of this token it will report an error. If *tag* is present, it declares a C type for *name*s as described for **%token**.

`%type` **[**<*tag*>**]** *name*`...`
: Declares that union member *name*s are non-terminals, and thus it is required to have a *tag* field at its beginning. Because it deals with non-terminals only, assigning a token number or using a literal is also prohibited. If this construct is present, *yacc* will perform type checking; if this construct is not present, the parse stack will hold only the **int** type.

Every name used in *grammar* undefined by a **%token**, **%left**, **%right** or **%nonassoc** declaration is assumed to represent a non-terminal symbol. The *yacc* utility will report an error for any non-terminal symbol that does not appear on the left side of at least one grammar rule.

Once the type, precedence or token number of a name is specified, it will not be changed. If the first declaration of a token does not assign a token number, *yacc* will assign a token number. Once this assignment is made, the token number will not be changed by explicit assignment.

The following declarators do not follow the previous pattern.

`%start` *name*
: Declares the non-terminal *name* to be the *start symbol*, which 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 can be overridden with this declaration.

`%union { body of union (in C) }`
: Declares the *yacc* value stack to be a union of the various types of values desired. By default, the values returned by actions (see below) and the lexical analyser will be integers. The *yacc* utility keeps track of types, and will insert corresponding union member names in order to perform strict type checking of the resulting parser.

  Alternatively, given that at least one <*tag*> construct is used, the union can be declared in a header file (which will be included in the declarations section by using an #include construct within **%{** and **%}**), and a **typedef** used to define the symbol YYSTYPE to represent this union. The effect of **%union** is to provide the declaration of YYSTYPE directly from the input.

`%{ ... %}`
: C-language declarations and definitions can appear in the declarations section, enclosed by these marks. These statements will be copied into the code file, and have global scope within it so that they can be used in the rules and program sections.

The declarations section must be terminated by the token **%%**.

Grammar Rules in yacc

The rules section defines the context-free grammar to be accepted by the function *yacc* generates, and associates with those rules C-language actions and additional precedence information. The grammar is described below, and a formal definition follows.

The rules section is comprised of one or more grammar rules. A grammar rule has the form:

```
A : BODY ;
```

The symbol A represents a non-terminal name, and **BODY** represents a sequence of zero or more *name*s, *literal*s and *semantic action*s that can then be followed by optional *precedence rule*s. Only the names and literals participate in the formation of the grammar; the semantic actions

and precedence rules are used in other ways. 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 appears only after the last rule. The BODY part can be empty (or empty of names and literals) to indicate that the non-terminal symbol matches the empty string.

The *yacc* utility assigns a unique number to each rule. Rules using the vertical bar notation are distinct rules. The number assigned to the rule appears in the description file.

The elements comprising a BODY are:

name
literal These form the rules of the grammar: *name* is either a *token* or a *non-terminal*; *literal* stands for itself (less the lexically required quotation marks).

semantic action
With each grammar rule, the user can associate actions to be performed each time the rule is recognised in the input process. (Note that the word ''action'' can also refer to the actions of the parser—shift, reduce, and so forth.)

These actions can return values and can obtain the values returned by previous actions. These values will be kept in objects of type YYSTYPE (see **%union**). The result value of the action will be kept on the parse stack with the left-hand side of the rule, to be accessed by other reductions as part of their right-hand side. By using the <*tag*> information provided in the declarations section, the code generated by *yacc* can be strictly type checked and contain arbitrary information. In addition, the lexical analyser can provide the same kinds of 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 C statements enclosed in curly braces { and }.

Certain pseudo-variables can be used in the action. These are macros for access to data structures known internally to *yacc*.

$$ The value of the action can be set by assigning it to $$. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

$*number* This refers to the value returned by the component specified by the token *number* in the right side of a rule, reading from left to right; *number* can be zero or negative. If it is, it refers to the data associated with the name on the parser's stack preceding the leftmost symbol of the current rule. (That is, $0 refers to the name immediately preceding the leftmost name in the current rule, to be found on the parser's stack and $–1 refers to the symbol to *its* left.) If *number* refers to an element past the current point in the rule, or beyond the bottom of the stack, the result is undefined. If type checking is enabled and the type of the value to be assigned cannot be determined, a diagnostic message may be generated.

$<*tag*>*number*
These correspond exactly to the corresponding symbols without the *tag* inclusion, but allow for strict type checking (and preclude unwanted type conversions). The effect is that the macro is expanded to use *tag* to select an element from the YYSTYPE union (using *dataname.tag*). This is particularly useful if *number* is not positive.

$**<*tag*>**$ This imposes on the reference the type of the union member referenced by *tag*. This construction is applicable when a reference to a left context value occurs in the grammar, and provides *yacc* with a means for selecting a type.

Actions can occur in the middle of a rule as well as at the end; an action can access values returned by actions to its left, and in turn the value it returns can be accessed by actions to its right. An action appearing in the middle of a rule will be equivalent to replacing the action with a new non-terminal symbol and adding an empty rule with that non-terminal symbol on the left-hand side. The semantic action associated with the new rule will be equivalent to the original action. The use of actions within rules might introduce conflicts that would not otherwise exist.

By default, the value of a rule is the value of the first element in it. If the first element does not have a type (particularly in the case of a literal) and type checking is turned on by **%type** an error message will result.

precedence

The keyword **%prec** can be used to change the precedence level associated with a particular grammar rule. Examples of this are in cases where a unary and binary operator have the same symbolic representation, but need to be given different precedences, or where the handling of an ambiguous if-else construction is necessary. The reserved symbol **%prec** can appear immediately after the body of the grammar rule and can be followed by a token name or a literal. It will cause the precedence of the grammar rule to become that of the following token name or literal. The action for the rule as a whole can follow **%prec**.

If a program section follows, the grammar rules must be terminated by **%%**.

Programs Section

The *programs* section can include the definition of the lexical analyser *yylex*(), and any other functions, for example those used in the actions specified in the grammar rules. This is C-language code, and will be included in the code file after the tables and code generated by *yacc*. It is unspecified whether the programs section precedes or follows the semantic actions in the output file; therefore, if the application contains any macro definitions and declarations intended to apply to the code in the semantic actions, it must place them within **%{...%}** in the declarations section.

Input Grammar

The following input to *yacc* yields a parser for the input to *yacc*. This formal syntax takes precedence over the preceding text syntax description.

The lexical structure is defined less precisely; **Lexical Structure of the Grammar** on page 829 defines most terms. The correspondence between the previous terms and the tokens below is as follows.

IDENTIFIER
This corresponds to the concept of *name*, given previously. It also includes literals as defined previously.

C_IDENTIFIER
This is a name, and additionally it is known to be followed by a colon. A literal cannot yield this token.

NUMBER
A string of digits (a non-negative decimal integer).

TYPE
LEFT
MARK
and so forth
These correspond directly to **%type**, **%left**, **%%** and so forth.

{ ... **}** This indicates C-language source code, with the possible inclusion of $ macros as discussed previously.

```
/*       Grammar for the input to yacc */

/*       Basic entries */

/*       The following are recognised by the lexical analyser */

%token   IDENTIFIER      /* includes identifiers and literals */
%token   C_IDENTIFIER    /* identifier (but not literal)
                            followed by a : */
%token   NUMBER          /* [0-9][0-9]* */

/*       Reserved words : %type=>TYPE %left=>LEFT, and so forth */

%token   LEFT RIGHT NONASSOC TOKEN PREC TYPE START UNION

%token   MARK            /* the %% mark */
%token   LCURL           /* the %{ mark */
%token   RCURL           /* the }% mark */

/*       8-bit character literals stand for themselves; */
/*       tokens have to be defined for multi-byte characters */

%start   spec


%%

spec  : defs MARK rules tail
      ;

tail  : MARK
      {
        /* In this action, set up the rest of the file */
```

```
        }
        | /* empty; the second MARK is optional */
        ;
defs    : /* empty */
        | defs def
        ;
def     : START IDENTIFIER
        | UNION
        {
          /* Copy union definition to output */
        }
        | LCURL
        {
          /* Copy C code to output file */
        }
          RCURL
        | rword tag nlist
        ;
rword   : TOKEN
        | LEFT
        | RIGHT
        | NONASSOC
        | TYPE
        ;
tag     : /* empty: union tag id optional */
        | '<' IDENTIFIER '>'
        ;
nlist   : nmno
        | nlist nmno
        ;
nmno    : IDENTIFIER        /* Note: literal invalid with % type */
        | IDENTIFIER NUMBER  /* Note: invalid with % type */
        ;
/*      rule section */
rules   : C_IDENTIFIER rbody prec
        | rules  rule
        ;
rule    : C_IDENTIFIER rbody prec
        | '|' rbody prec
        ;
rbody   : /* empty */
        | rbody IDENTIFIER
        | rbody act
```

```
        ;
act     : '{'
          {
                  /* Copy action, translate $$, and so forth */
          }
          '}'
        ;
prec    : /* empty */
        | PREC IDENTIFIER
        | PREC IDENTIFIER act
        | prec ';'
        ;
```

Conflicts

The parser produced for an input grammar may contain states in which conflicts occur. The conflicts occur because the grammar is not LALR(1). An ambiguous grammar always contains at least one LALR(1) conflict. The *yacc* utility will resolve all conflicts, using either default rules or user-specified precedence rules.

Conflicts are either shift/reduce conflicts or reduce/reduce conflicts. A shift/reduce conflict is where, for a given state and lookahead symbol, both a shift action and a reduce action are possible. A reduce/reduce conflict is where, for a given state and lookahead symbol, reductions by two different rules are possible.

The rules below describe how to specify what actions to take when a conflict occurs. Not all shift/reduce conflicts can be successfully resolved this way because the conflict may be due to something other than ambiguity, so incautious use of these facilities can cause the language accepted by the parser to be much different from that which was intended. The description file will contain sufficient information to understand the cause of the conflict. Where ambiguity is the reason either the default or explicit rules should be adequate to produce a working parser.

The declared precedences and associativities (see **Declarations Section** on page 829) 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 might not have both precedence and associativity.
2. 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 favour 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 non-associative implies an error in the string being parsed.
3. When there is a shift/reduce conflict that cannot be resolved by rule 2, the shift is done. Conflicts resolved this way are counted in the diagnostic output described in **Error Handling** on page 836.
4. When there is a reduce/reduce conflict, a reduction is done by the grammar rule that occurs earlier in the input sequence. Conflicts resolved this way are counted in the diagnostic output described in **Error Handling** on page 836.

Conflicts resolved by precedence or associativity will not be counted in the shift/reduce and reduce/reduce conflicts reported by *yacc* on either standard error or in the description file.

Error Handling

The token **error** is reserved for error handling. The name **error** can be used in grammar rules. It indicates places where the parser can recover from a syntax error. The default value of **error** is 256. Its value can be changed using a **%token** declaration. The lexical analyser should not return the value of **error**. (Multi-byte characters should be recognised by the lexical analyser and returned as tokens. They should not be returned as multi-byte character literals. The token **error** that is used for error recovery is normally assigned the value 256 in the historical implementation. Thus, the token value 256, which used in many multi-byte character sets, is not available for use as the value of a user-defined token.)

The parser will detect a syntax error when it is in a state where the action associated with the lookahead symbol is **error**. A semantic action can cause the parser to initiate error handling by executing the macro YYERROR. When YYERROR is executed, the semantic action will pass control back to the parser. YYERROR cannot be used outside of semantic actions.

When the parser detects a syntax error, it normally calls **yyerror** with the character string "syntax error" as its argument. The call will not be made if the parser is still recovering from a previous error when the error is detected. The parser is considered to be recovering from a previous error until the parser has shifted over at least three normal input symbols since the last error was detected or a semantic action has executed the macro **yyerrok**. The parser will not call **yyerror** when YYERROR is executed.

The macro function YYRECOVERING() will return 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero will be returned.

When a syntax error is detected by the parser, the parser will check if a previous syntax error has been detected. If a previous error was detected, and if no normal input symbols have been shifted since the preceding error was detected, the parser checks if the lookahead symbol is an endmarker (see **Interface to the Lexical Analyser** on page 837). If it is, the parser will return with a non-zero value. Otherwise, the lookahead symbol will be discarded and normal parsing will resume.

When YYERROR is executed or when the parser detects a syntax error and no previous error has been detected, or at least one normal input symbol has been shifted since the previous error was detected, the parser will pop back one state at a time until the parse stack is empty or the current state allows a shift over **error**. If the parser empties the parse stack, it will return with a non-zero value. Otherwise, it will shift over **error** and then resume normal parsing. If the parser reads a lookahead symbol before the error was detected, that symbol will still be the lookahead symbol when parsing is resumed.

The macro **yyerrok** in a semantic action will cause the parser to act as if it has fully recovered from any previous errors. The macro **yyclearin** will cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, **yyclearin** will have no effect.

The macro YYACCEPT will cause the parser to return with the value zero. The macro YYABORT will cause the parser to return with a non-zero value.

Interface to the Lexical Analyser

The *yylex*( ) function is an integer-valued function that returns a *token number* representing the kind of token read. If there is a value associated with the token returned by *yylex*( ) (see the discussion of *tag* above), it will be assigned to the external variable *yylval*.

If the parser and *yylex*( ) do not agree on these token numbers, reliable communication between them cannot occur. For (one character) literals, the token is simply the numeric value of the character in the current character set. The numbers for other tokens can either 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. The **#define** statements are put into the code file, and the header file if that file is requested. The set of characters permitted by *yacc* in an identifier is larger than that permitted by C. Token names found to contain such characters will not be included in the **#define** declarations.

If the token numbers are chosen by *yacc*, the tokens other than literals will be assigned numbers greater than 256, although no order is implied. A token can be explicitly assigned a number by following its first appearance in the declarations section with a number. Names and literals not defined this way retain their default definition. All assigned token numbers will be unique and distinct from the token numbers used for literals. If duplicate token numbers cause conflicts in parser generation, *yacc* will report an error; otherwise, it is unspecified whether the token assignment is accepted or an error is reported.

The end of the input is marked by a special token called the *endmarker*, which has a token number that is zero or negative. (These values are invalid for any other token.) All lexical analysers will return zero or negative as a token number upon reaching the end of their input. If the tokens up to, but excluding, the endmarker form a structure that matches the start symbol, the parser will accept the input. If the endmarker is seen in any other context, it will be considered an error.

Completing the Program

In addition to *yyparse*( ) and *yylex*( ), the functions *yyerror*( ) and *main*( ) are required to make a complete program. The application can supply *main*( ) and *yyerror*( ), or those routines can be obtained from the *yacc* library.

Yacc Library

The following functions appear only in the *yacc* library accessible through the **–l y** operand to *cc* or *c89*; they can therefore be redefined by a portable application:

int main(void)

This function will call *yyparse*( ) and exit with an unspecified value. Other actions within this function are unspecified.

int yyerror(const char *s)

This function will write the NUL-terminated argument to standard error, followed by a newline character.

The order of the **–l y** and **–l l** operands given to *cc* or *c89* is significant; the application must either provide its own *main*( ) function or ensure that **–l y** precedes **–l l**.

Debugging the Parser

The parser generated by *yacc* will have diagnostic facilities in it that can be optionally enabled at either compile time or at run time (if enabled at compile time). The compilation of the runtime debugging code is under the control of YYDEBUG, a preprocessor symbol. If YYDEBUG has a non-zero value, the debugging code will be included. If its value is zero, the code will not be included.

In parsers where the debugging code has been included, the external int yydebug can be used to turn debugging on (with a non-zero value) and off (zero value) at run time. The initial value of *yydebug* will be zero.

When **–t** is specified, the code file will be built such that, if YYDEBUG is not already defined at compilation time (using the *c89* **–D YYDEBUG** option, for example), YYDEBUG will be set explicitly to 1. When **–t** is not specified, the code file will be built such that, if YYDEBUG is not already defined, it will be set explicitly to zero.

The format of the debugging output is unspecified but includes at least enough information to determine the shift and reduce actions, and the input symbols. It also provides information about error recovery.

Algorithms

The parser constructed by *yacc* implements an LALR(1) parsing algorithm as documented in the literature. It is unspecified whether the parser is table-driven or direct-coded.

A parser generated by **yacc** will never request an input symbol from *yylex*( ) while in a state where the only actions other than the error action are reductions by a single rule.

The literature of parsing theory defines these concepts.

Limits

The *yacc* utility may have several internal tables. The minimum maximums for these tables are shown in the following table. The exact meaning of these values is implementation-dependent. The implementation will define the relationship between these values and between them and any error messages that the implementation may generate should it run out of space for any internal structure. An implementation may combine groups of these resources into a single pool as long as the total available to the user does not fall below the sum of the sizes specified by this section.

Limit	Minimum Maximum	Description
{NTERMS}	126	Number of tokens.
{NNONTERM}	200	Number of non-terminals.
{NPROD}	300	Number of rules.
{NSTATES}	600	Number of states.
{MEMSIZE}	5200	Length of rules. The total length, in names (tokens and non-terminals), of all the rules of the grammar. The left-hand side is counted for each rule, even if it is not explicitly repeated, as specified in **Grammar Rules in yacc** on page 830.
{ACTSIZE}	4000	Number of actions. ''Actions'' here (and in the description file) refer to parser actions (shift, reduce, and so forth) not to semantic actions defined in **Grammar Rules in yacc** on page 830.

Table 3-17 Internal Limits in *yacc*

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

If any errors are encountered, the run is aborted and *yacc* exits with a non-zero status. Partial code files and header files files may be produced. The summary information in the description file will always be produced if the **−v** flag is present.

APPLICATION USAGE

Historical implementations experience name conflicts on the names yacc.tmp, yacc.acts, yacc.debug, y.tab.c, y.tab.h and y.output if more than one copy of *yacc* is running in a single directory at one time. The **−b** option was added to overcome this problem. The related problem of allowing multiple *yacc* parsers to be placed in the same file was addressed by adding a **−p** option to override the previously hard-coded yy variable prefix.

The description of the **−p** option specifies the minimal set of function and variable names that cause conflict when multiple parsers are linked together. YYSTYPE does not need to be changed. Instead, the programmer can use **−b** to give the header files for different parsers different names, and then the file with the *yylex*( ) for a given parser can include the header for that parser. Names such as *yyclearerr* do not need to be changed because they are used only in the actions; they do not have linkage. It is possible that an implementation will have other names, either internal ones for implementing things such as *yyclearerr*, or providing non-standard features that it wants to change with **−p**.

Unary operators that are the same token as a binary operator in general need their precedence adjusted. This is handled by the **%prec** advisory symbol associated with the particular grammar rule defining that unary operator. See **Grammar Rules in yacc** on page 830. Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

EXAMPLES

Access to the *yacc* library is obtained with library search operands to *cc* or *c89*. To use the *yacc* library *main*( ):

```
c89 y.tab.c -l y
```

Both the *lex* library and the *yacc* library contain *main*( ). To access the *yacc main*( ):

```
c89 y.tab.c lex.yy.c -l y -l l
```

This ensures that the *yacc* library is searched first, so that its *main*( ) is used.

The historical *yacc* libraries have contained two simple functions that are normally coded by the application programmer. These library functions are similar to the following code:

```
#include <locale.h>
int main(void)
{
       extern int yyparse();

       setlocale(LC_ALL, "");

       /* If the following parser is one created by lex, the
          application must be careful to ensure that LC_CTYPE
          and LC_COLLATE are set to the POSIX locale.  */
       (void) yyparse();
       return (0);
}

#include <stdio.h>

int yyerror(const char *msg)
{
       (void) fprintf(stderr, "%s\n", msg);
       return (0);
}
```

FUTURE DIRECTIONS

None.

SEE ALSO

cc, *c89*, *lex*.

CHANGE HISTORY

First released in Issue 2.

Issue 3

The operation of this utility in an 8-bit transparent manner has been noted.

The operation of this utility in an internationalised environment has been described.

Issue 4

Aligned with the ISO / IEC 9945-2: 1993 standard.

NAME

zcat – expand and concatenate data

SYNOPSIS

EX `zcat [file...]`

DESCRIPTION

The *zcat* utility will write to standard output the uncompressed form of files that have been compressed using the *compress* utility. It is the equivalent of *uncompress* –**c**. Input files are not affected.

OPTIONS

None.

OPERANDS

The following operand is supported:

file The pathname of a file previously processed by the *compress* utility. If *file* already has the .Z suffix specified, it is used as submitted. Otherwise, the .Z suffix is appended to the filename prior to processing.

STDIN

The standard input will be used only if no *file* operands are specified, or if a *file* operand is –.

INPUT FILES

Input files must be compressed files that are in the format produced by the *compress* utility.

ENVIRONMENT VARIABLES

The following environment variables affect the execution of *zcat*:

LANG Provide a default value for the internationalisation variables that are unset or null. If *LANG* is unset or null, the corresponding value from the implementation-specific default locale will be used. If any of the internationalisation variables contains an invalid setting, the utility will behave as if none of the variables had been defined.

LC_ALL

If set to a non-empty string value, override the values of all the other internationalisation variables.

LC_CTYPE

Determine the locale for the interpretation of sequences of bytes of text data as characters (for example, single- as opposed to multi-byte characters in arguments).

LC_MESSAGES

Determine the locale that should be used to affect the format and contents of diagnostic messages written to standard error.

NLSPATH

Determine the location of message catalogues for the processing of *LC_MESSAGES*.

ASYNCHRONOUS EVENTS

Default.

STDOUT

The compressed files given as input will be written on standard output in their uncompressed form.

STDERR

Used only for diagnostic messages.

OUTPUT FILES

None.

EXTENDED DESCRIPTION

None.

EXIT STATUS

The following exit values are returned:

0 Successful completion.
>0 An error occurred.

CONSEQUENCES OF ERRORS

Default.

APPLICATION USAGE

None.

EXAMPLES

None.

FUTURE DIRECTIONS

None.

SEE ALSO

compress, *pcat*, *uncompress*, *zcat*.

CHANGE HISTORY

First released in Issue 4.

Index