-- --

IEEE P1003.2 Draft 11.2 – September 1991

P1003.2 Draft 11.2
ISO/IEC CD 9945-2.2

STANDARDS PROJECT

Draft Standard for Information Technology — Portable Operating System Interface (POSIX) Part 2: Shell and Utilities

Sponsor
Technical Committee on Operating Systems and Application Environments
of the
IEEE Computer Society

Work Item Number: JTC 1.22.21.2

Abstract: ISO/IEC 9945-2: 199x (IEEE Std 1003.2-199x) is part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to a shell command language and a set of utility programs for complex data manipulation.

Keywords: API, application portability, data processing, open systems, operating system, portable application, POSIX, shell and utilities

P1003.2 / D11.2
September 1991

IEEE Standards Department
Copyright and Permissions
445 Hoes Lane, P.O. Box 1331
Piscataway, NJ 08855-1331, USA
+1 (908) 562-3800
+1 (908) 562-1571 [FAX]
September 1991 *SH XXXXX*

Editor's Notes

The IEEE ballot for Draft 11.2 is due at the IEEE Standards Office on 2
21 October 1991. You are also asked to e-mail any balloting com- 2
ments to me: `hlj@posix.com`. Please read the balloting instruc- 2
tions in Annex G. 2

This document is also registered as ISO/IEC CD 9945-2.2. The international bal- 2
loting period is unrelated to the IEEE balloting. Member bodies, please consult 2
any accompanying materials from SC22. Also, please read the remainder of these 2
Editor Notes to see explanations of stylistic differences between a draft and the 2
final standard (copyright notices, inline rationale, etc.). 2

The IEEE balloting will be on hiatus during the international balloting period, 2
which is probably scheduled to complete at the May 1992 WG15 meeting. This is 2
in accordance with the WG15 Synchronization Plan, which calls for coordinated 2
balloting to result in the approval of an IEEE/ANSI standard that is identical to 2
the ISO/IEC Draft International Standard (DIS). There will be a final recircula- 2
tion of a full draft (12) to the IEEE balloting group before it is sent to the Stan- 2
dards Board. 2

This section will not appear in the final document. It is used for editorial com- 2
ments concerning this draft. Draft 11.2 is the fifth recirculation of the balloting 2
process that began in December 1988 with Draft 8. Please consult Annex G and
the cover letter for the ballot that accompanied this draft for information on how
the recirculation is accomplished.

This draft uses small numbers in the right margin in lieu of change bars. ''2'' 2
denotes changes from Draft 11.1 to Draft 11.2. ''1'' denotes changes from Draft 11 2
to Draft 11.1. All diff-marks prior to Draft 11.1 have been removed. Trivial infor- 1
mative (i.e., non-normative) changes and purely editorial changes such as gram-
mar, spelling, or cross references are not diff-marked.

There are two versions of Draft 11.2 in circulation. The full printed version was 2
sent for SC22 balloting and is also available from the IEEE for a duplication fee 2
[call (800) 678-IEEE or +1 (908) 981-1393 outside the US]. The version sent to the 2
IEEE balloting group consists (mostly) of pages containing normative changes. 1
This was done to focus balloting group attention on the changes being balloted 1
and to reduce costs and administrative time. The changes-only version contains a 1
few handwritten pointers in the margins to show context where it would not be 1
obvious; numbers near the normal page numbers show what the corresponding 1
Draft 11 page number would be. 1

The following minor global changes have been made without diff-marks:

— Instances of the verbs ''print,'' ''report,'' ''display,'' ''issue,'' and ''list'' are being changed to ''write'' as part of a general cleanup related to the UPE, where ''write'' and ''display'' have precise meanings. This is probably not completed and will continue throughout ballot resolution and the final editing process.

ISO and IEEE have tightened up the requirements for the use of ‘‘shall.’’ We have been directed that all sentences that are currently declarative must be changed to use the ‘‘shall’’ form if they pose a requirement: ‘‘The status is zero’’ → ‘‘The status shall be zero.’’ One specific instance of this was changing ‘‘The following options/operands are available’’ to ‘‘The following options/operands shall be supported by the implementation.’’ Another: ‘‘The `foo` utility follows the utility argument syntax standard described in 2.11.2’’ to ‘‘The `foo` utility shall conform to the utility argument syntax guidelines described in 2.10.2.’’ It is a tedious process to do all these translations and they are not complete. They will completed on a draft-by-draft basis. In the meantime, please assume that all declarative sentences mean to use ‘‘shall’’ and treat them as either implementation or application requirements unless they specifically say ‘‘may,’’ ‘‘should,’’ or ‘‘can.’’

The rationale text for all the sections has been temporarily moved from Annex E and interspersed with the appropriate sections. The rationale sections are identified with the phrase ‘‘*(This subclause is not a part of P1003.2)*’’ in the heading. This colocation of rationale with its accompanying text was done to encourage the Technical Reviewers to maintain the rationale text, as well as provide explanations to the reviewers and balloters. Not all of the Rationale sections have contents as of this draft. The empty sections may be partially distracting, but we feel it is imperative to keep them there to encourage the Technical Reviewers to provide rationale as needed.

Please report typographical errors to:

Hal Jespersen
POSIX Software Group
447 Lakeview Way
Redwood City, CA 94062
+1 (415) 364-3410
FAX: +1 (415) 364-4498
Email: `hlj@Posix.COM`

(Electronic mail is preferred.)

The copying and distribution of IEEE balloting drafts is accomplished by the Standards Office. To report problems with reproduction of your copy, contact: 2

Anna Kaczmarek 2
IEEE Standards Office
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331

+1 (908) 562-3811 2
FAX: +1 (908) 562-1571

Additional copies of this draft are available for a duplication and mailing fee. Contact:

IEEE Publications
1 (800) 678-IEEE
+1 (908) 981-1393 [outside US]

This draft is available in various electronic forms to assist the review process. Our thanks to Andrew Hume of AT&T Bell Laboratories for providing online access facilities. Note that this is a limited experiment in providing online access; future ballots may provide other forms, such as diskettes or a bulletin board arrangement, but the instructions shown here are the only methods currently available. Please also observe the additional copyright restrictions that are described in the online files.

Assuming you have access to the Internet, the scenario is approximately

```
ftp research.att.com # research's IP address is 192.20.225.2
<login as netlib; password is your email address>
cd posix/p1003.2/d11.2
get toc index
binary
get p11-20.Z
```

The draft is available in several forms. The table of contents can be found in `toc`, pages containing a particular section are stored under the section number, sets of pages are stored in files with names of the form `p`*n*−*m*, and the entire draft is stored in `all`. By default, files are ASCII. A `.ps` suffix indicates PostScript. A `.Z` suffix indicates a `compress`'ed file. The file `index` contains a general description of the files available.

These files are also available via electronic mail by sending a message like

```
send 3.4 3.5 9.2 from posix/p1003.2/d11.2
```

to `netlib@research.att.com`. If you use email, you should *not* ask for the compressed version. For a more complete introduction to this form of *netlib*, send the message

```
send help
```

POSIX.2 Change History

This section is provided to track major changes between drafts. Since it was first added in Draft 11, earlier entries omit some degree of detail.

Draft 11.2 [September 1991] Sixth IEEE ballot (fifth recirculation; only 2
changed pages distributed). Second ISO/IEC CD 9945-2 registra- 2
tion (full draft distributed). 2

- Equivalence classes as starting/ending points of regular 2
expression bracket expression range expression have been 2
made unspecified. 2
- The LC_COLLATE `substitute` keyword has been deleted. 2
- `cksum` (4.9): Modifications to the algorithm. 2
- `cp` (4.13): Restoration of the 2
- `stty` (4.59): Addition of the `tostop` operand. 2
- `lex` (A.2): Further clarification of ERE differences. 2
- Miscellaneous clarifications to various utilities. 2

Draft 11.1 [June 1991] Fifth IEEE ballot (fourth recirculation; only changed 1
pages distributed). 1

- Modification of the definition of *byte* and clarifications of 1
octal/hexadecimal byte representations throughout the utili- 1
ties. 1
- Clarifications to the locale definition source file description in 1
2.5; addition of a `yacc` grammar. 1
- Removal of `pax −e` character translation option. 1
- Miscellaneous clarifications to various utilities. 1
- Reconciliation of feature test macros and headers in Annex B 1
with POSIX.1. 1

Draft 11 [February 1991] Fourth IEEE ballot (third recirculation).

- Changes in 2.3 to the treatment of regular built-ins in regards to their *exec*-able versions.
- Changes to 2.4 (character names and charmap syntax) and 2.5 (`localedef` input format) as a result of international balloting. Addition of the {POSIX2_LOCALEDEF} symbol.
- Changes to the shell quoting rules, arithmetic expression syntax, command search order, error descriptions, and exportable functions.
- Movement of the `command` utility from special built-in status to be a utility in Section 4.

— `cp` (4.13): Significant clarifications and interface changes.

— `date` (4.15): Added field descriptor modifiers to handle alternate calendar forms when supported by the locale and implementation.

— `pax` (4.48): Significant interface changes, including international character set translations.

— `test` (4.62): Deprecated some functionality due to inconsistent behavior in existing implementations that cause portability problems in existing applications.

— `make` (6.2): Addition of the `.POSIX` special target, return of some rules to strict existing practice.

— Miscellaneous clarifications to various utilities.

— The FORTRAN section now has two options associated with it: Development Utilities (`fort77`) and Runtime Utilities (`asa`).

— Addition of full example profiles and charmaps from Denmark in Annex F.

Draft 10 [July 1990] Third IEEE ballot (second recirculation).

— This draft primarily has been one of clarification and amplification. In resolving ballot objections, large portions of the draft have been rewritten, affecting all sections, but comparatively few changes in [intended] functionality have occurred.

— New shell command language features (see Section 3):

— Utility name changes:

Draft 9	Draft 10
`create`	`pathchk`
`hexdump`	`od`
`sendto`	`mailx`

— A few of the utilities and global sections now have a more formal description, using a `yacc`-like grammar.

— Considerably more detail has been added to the internationalization features of the standard: global changes to clauses 2.4 and 2.5; new detail to the LC_* variables in each utility section; specification of **LC_MESSAGES** (replacing **LC_RESPONSE**).

— Due to some ISO requirements, Sections 1 and 2 have been reorganized yet again, causing many cross reference number changes. The Related Standards annex has been turned into simply a Bibliography. The Non-Specified Language Compilers annex has been replaced by a Sample National Profile annex.

Draft 9	[August 1989] Second IEEE ballot (first recirculation). Also registered as ISO/IEC CD 9945-2.1. A few minor corrections to some sections. :-)
Draft 8	[December 1988] First IEEE ballot. Also submitted to ISO/IEC JTC 1/SC22 for review and comment.
Draft 7	[September 1988] ‘‘Mock ballot’’ conducted by working group members only.

POSIX.2 Technical Reviewers

The individuals denoted in Table i are the Technical Reviewers for this draft. During balloting they are the subject matter experts who coordinate the resolution process for specific sections, as shown.

Table i — POSIX.2 Technical Reviewers

Section	Description	Reviewer	
1	*General*	Jespersen	
2.4,2.5	*Definitions (Locales)*	Leijonhufvud	1
2 (rest)	*Definitions (Various)*	Jespersen	
3	*Command Language*	Jespersen	
4	*Execution Environment Utilities:* `cp`, `rm`	Bostic	2
4	*Execution Environment Utilities: (the rest)*	Jespersen	2
6	*Software Development Utilities*	Jespersen	
7	*Language-Independent Bindings*	Jespersen	2
A	*C Development Utilities*	Jespersen	
B	*C Bindings*	Jespersen	2
C	*FORTRAN Development and Runtime Utilities*	Jespersen	
D-G	*Various*	Jespersen	

Also, our special thanks to Donn Terry for writing or improving all the `yacc`-based grammars used in Draft 10.

POSIX.2 Proposed Schedule

This section will not appear in the final document. It is used to provide editorial notes regarding the proposed POSIX.2 schedule. In the schedule, the UPE stands for ‘‘User Portability Extension.’’

Date	Milestone (End of Meeting)	Draft	
Sep 7-11, 1987 Nashua, NH	Utility format frozen; 10% of utilities described.	**3**	
Dec 7-14, 87 San Diego, CA	50% of utilities described; shell update; substantial progress in Sections 2, 3, 4, 8.	**4**	
Mar 14-18, 1988 Washington, DC	Utility selection frozen; 75% described.	**5**	
Jul 11-15, 1988 Denver, CO	100% utilities described; functional freeze; produce ‘‘mock ballot’’ and POSIX FIPS draft 7	**6**	
[Sep-Oct 1988]	[Mock ballot]	**7**	
Oct 24-28, 1988 Honolulu, HI	Resolve mock ballot objections; produce first real ballot (draft 8) UPE planning begins	**7**	
[Jan-Feb 1989]	[First ballot]	**8**	
Jan 9-11, 1989 Ft. Lauderdale, FL	Begin UPE definitions; Technical Reviewer coordination of first ballot responses	**8**	
[Feb-Apr 1989]	[Ballot resolution]	**8**	
Apr 24-28, 1989 Minneapolis, MN	Working Group concurrence with ballot resolution; produce Draft 9 for recirculation; UPE work	**9**	
Jul 10-14, 1989 San Jose, CA	UPE work		
[Oct 1989]	[First Recirculation]	**9**	
[Nov-Feb 1990]	[Ballot resolution]	**9**	
[Aug-Sep 1990]	[Second Recirculation]	**10**	
[Mar 1991]	[Third Recirculation]	**11**	
[Jun 1991]	[Fourth Recirculation]	**11.1**	1
[Sep 1991]	[Fifth Recirculation]	**11.2**	1
[mid-1992]	[IEEE Standard Board Approves??]	**12**	2
[Jul 1990 - Apr 1992]	[Ballot .2a UPE supplement]		1

IEEE Standards documents are developed within the Technical Committees of the IEEE Societies and the Standards Coordinating Committees of the IEEE Standards Board. Members of the committees serve voluntarily and without compensation. They are not necessarily members of the Institute. The standards developed within IEEE represent a consensus of the broad expertise on the subject within the Institute as well as those activities outside of IEEE that have expressed an interest in participating in the development of the standard.

Use of an IEEE Standard is wholly voluntary. The existence of an IEEE Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of the IEEE Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change brought about through developments in the state of the art and comments received from users of the standard. Every IEEE Standard is subjected to review at least every five years for revision or reaffirmation. When a document is more than five years old and has not been reaffirmed, it is reasonable to conclude that its contents, although still of some value, do not wholly reflect the present state of the art. Users are cautioned to check to determine that they have the latest edition of any IEEE Standard.

Comments for revision of IEEE Standards are welcome from any interested party, regardless of membership affiliation with IEEE. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of the IEEE, the Institute will initiate action to prepare appropriate responses. Since IEEE Standards represent a consensus of all concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, the IEEE and the members of its technical committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments on standards and requests for interpretations should be addressed to:

Secretary, IEEE Standards Board
445 Hoes Lane
P.O. Box 1331
Piscataway, NJ 08855-1331

IEEE Standards documents are adopted by the Institute of Electrical and Electronics Engineers without regard to whether their adoption may involve patents on articles, materials, or processes. Such adoption does not assume any liability to any patent owner, nor does it assume any obligation whatever to parties adopting the standards documents.

Contents

PAGE

PAGE

FIGURES

TABLES

Introduction

(This Introduction is not a normative part of P1003.2 Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities, but is included for information only.)

The purpose of this standard is to define a standard interface and environment for application programs that require the services of a ''shell'' command language interpreter and a set of common utility programs. It is intended for systems implementors and application software developers, and is complementary to ISO/IEC 9945-1: 1990 {8} (first in a family of 'POSIX'' standards), which specifies operating system interfaces and source code level functions, based on the UNIX[1)] system documentation. This standard, or 'POSIX. 2,'' is based upon documentation and the knowledge of existing programs that assume an interface and architecture similar to that described by POSIX. 1. (See 1.1 for a full description of the relationship between the standards.)

The majority of this standard describes the functions of utilities that can interface with application programs. The standard also provides high-level language interfaces that the application uses to access these utilities and other useful, related services. These language-independent service interfaces are temporarily described in terms of their C language bindings. The C language assumed is that defined by the C Standard: *ANSI /X3.159-1989 Programming Language C Standard* produced by Technical Committee X3J11 of the Accredited Standards Committee X3 — Information Processing Systems.

Organization of the Standard

The standard is divided into ten parts:

— General, including a statement of scope, normative references, and conformance requirements. (Section 1).

— Definitions, general requirements, and the environment available to applications. (Section 2).

— The shell command interpreter language. (Section 3).

— Descriptions of the utilities in the required ''Execution Environment Utilities.'' (Section 4).

— Descriptions of the utilities required for user portability on asynchronous terminals. (Section 5 [to be provided in a future revision]).

— Descriptions of the utilities in the optional ''Software Development Utilities.'' (Section 6).

---

1) UNIX is a registered trademark of UNIX System Laboratories in the USA and other countries.

— Language-independent interfaces for high-level programming language access to shell and related services. (Section 7).

— Descriptions of the utilities in the optional ‘‘C Language Development Utilities.’’ (Normative Annex A).

— C language bindings to the interfaces in Section 6. (Normative Annex B).

— Descriptions of the utilities in the optional ‘FORTRAN Development and Runtime Utilities.’’ (Normative Annex C).

This introduction, the foreword, any footnotes, NOTES accompanying the text, and the *informative* annexes are not considered part of the standard. Annexes D through G are informative.

Base Documents

Many of the interfaces and utilities of this standard were adapted from materials in machine-readable forms donated by the following organizations:

— AT&T: the *System V Interface Definition* (*SVID*) {B24},[2] Issue 2, Volume 2. Copyright © 1986, AT&T; reprinted with permission.

— The X/Open Company, Ltd.: the *X/Open Portability Guide* {B30} {B31}, Issues II and III, Volume 1. Copyright © 1989, X/Open Company, Ltd; reprinted with permission.

— University of California, *The UNIX User's Reference Manual* {B28}, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version, 1986. Copyright © 1980, 1983, The Regents of the University of California; reprinted with permission. [3]

Significant reference use was also made of the following books:

— Bolsky, Morris I., Korn, David G., *The KornShell Command and Programming Language* {B25}, Prentice Hall, Englewood Cliffs, New Jersey (1988).

— Aho, Alfred V., Kernighan, Brian W., Weinberger, Peter J., *The AWK Programming Language* {B21}, Addison-Wesley, Reading, Massachusetts (1988).

Many other proposals for functions and utilities were received from the various working group members, who are listed in the Acknowledgements section of this standard.

---

2) The number in braces corresponds to those of the references in 1.2 (or the bibliographic entry in Annex D if the number is preceded by the letter B).

3) The IEEE is grateful to AT&T, UniForum, and the Regents of the University of California for permission to use their machine-readable materials.

Related Standards Activities

Activities to extend this standard to address additional requirements are in progress, and similar efforts can be anticipated in the future.

The following areas are under active consideration at this time, or are expected to become active in the near future: [4]

(1) Language-independent service descriptions of POSIX.1 {8}

(2) C, Ada, and FORTRAN Language bindings to (1)

(3) Verification testing methods

(4) Realtime facilities

(5) Secure/Trusted System considerations

(6) Network interface facilities

(7) System Administration

(8) Graphical User Interfaces

(9) Profiles describing application- or user-specific combinations of Open Systems standards for: supercomputing, multiprocessor, and batch extensions; transaction processing; realtime systems; and multiuser systems based on historical models

(10) An overall guide to POSIX-based or related Open Systems standards and profiles

Extensions are approved as ‘‘amendments’’ or ‘‘revisions’’ to this document, following the IEEE and ISO/IEC Procedures.

Approved amendments are published separately until the full document is reprinted and such amendments are incorporated in their proper positions.

If you have interest in participating in the TCOS working groups addressing these issues, please send your name, address, and phone number to the Secretary, IEEE Standards Board, Institute of Electrical and Electronics Engineers, Inc., P.O. Box 1331, 445 Hoes Lane, Piscataway, NJ 08855-1331, and ask to have this forwarded to the chairperson of the appropriate TCOS working group. If you have interest in participating in this work at the international level, contact your ISO/IEC national body.

---

4) A *Standards Status Report* that lists all current IEEE Computer Society standards projects is available from the IEEE Computer Society, 1730 Massachusetts Avenue NW, Washington, DC 20036-1903; Telephone: +1 202 371-0101; FAX: +1 202 728-9614. Working drafts of POSIX standards under development are also available from this office.

P1003.2 was prepared by the 1003.2 working group, sponsored by the Technical Committee on Operating Systems and Application Environments of the IEEE Computer Society. At the time this standard was approved, the membership of the 1003.2 working group was as follows:

Technical Committee on Operating Systems and Application Environments (TCOS)

Chair: Jehan-François Pâris

TCOS Standards Subcommittee

Chair:	Jim Isaak
Vice Chairs:	Ralph Barker
	David Dodge
	Robert Bismuth
	Hal Jespersen
	Lorraine Kevra
Treasurer:	Quin Hahn
Secretary:	Shane McCarron

1003.2 Working Group Officials

Chair:	Hal Jespersen
Vice Chair:	Donald W. Cragun
Editors:	Hal Jespersen (1986, 1988-1991)
	Maggie Lee (1987-1988)
Secretaries:	Helene Armitage (1988-1990)
	Dave Grindeland (1991)
	Robert J. Makowski (1987-1988)

Technical Reviewers

Helene Armitage	Ken Faubel	Gary Miller
Keith Bostic	Greger Leijonhufvud	Marc Teitelbaum
John Caywood	Bob Lenk	Donn Terry
Donald Cragun	Mark Levine	Teoman Topcubasi
David Decot	Shane McCarron	David Willcox

Working Group

Helene Armitage	Quin Hahn	Jim Oldroyd
Brian Baird	Michael J. Hannah	Mark Parenti
John R. Barr	Marjorie E. Harris	John Peace
Philippe Bertrand	David F. Hinnant	Jon Penner
Robert Bismuth	Leon M. Holmes	Gerald Powell
Jim Blondeau	Ron Holt	John Quarterman
James C. Bohem	Randall Howard	Joe Ramus
Kathy Bohrer	Steven A. James	Mike Ressler
Keith Bostic	Steve Jennings	Grover Righter
Phyllis Eve Bregman	Hal Jespersen	Andrew K. Roach
Peter Brouwer	Ronald S. Karr	Marco P. Roodzant
F. Lee Brown, Jr.	Lorraine C. Kevra	Seth Rosenthal
Jonathan Brown	Martin Kirk	Maude Sawyer

James A. Capps	Brad Kline	Norman K. Scherer
Bill Carpenter	Hiromichi Kogure	Glen Seeds
Steve Carter	David Korn	Jim Selkaitis
John Caywood	Rick Kuhn	Karen Sheaffer
Bob Claeson	Mike Lambert	Del Shoemaker
Mark Colburn	Maggie Lee	James Soddy
Donald W. Cragun	Perry Lee	Daniel Steinberg
Dave Decot	Greger Leijonhufvud	Scott A. Sutter
Terence S. Dowling	Bob Lenk	Ravi Tavakley
Stephen Dum	Mark Levine	Marc Teitelbaum
Dominic Dunlop	Gary Lindgren	Donn Terry
Mike Edmonds	John Lomas	Jack Thompson
Ron Elliott	Craig Lund	Teoman Topcubasi
Richard W. Elwood	Rod MacDonald	Eugene Tsuno
Hirsaki Eto	Dan Magenheimer	Geraldine Vitovitch
Fran Fadden	Robert J. Makowski	Carl vonLoewenfeldt
Ken Faubel	Shane P. McCarron	Mike Wallace
Martin C. Fong	Jim McGinness	Alan Weaver
Terance Fong	John McGrory	Larry Wehr
Glenn Fowler	Stuart McKaig	Bruce Weiner
Gary A. Gaudet	Sunil Mehta	N. Ray Wilkes
Al Gettier	Bill Middlecamp	David Willcox
Timothy D. Gill	Gary W. Miller	Neil Winton
Gregory Goddard	Jim Moe	David Woodend
Loretta Goudie	Yasushi Nakahara	Morten With
Dave Grindeland	Martha Nalebuff	Ken Witte
John Lawrence Gregg	Sonya D. Neufer	John Wu
Jerry Gross	Landon Noll	Peggy Younger
Douglas A. Gwyn	Robin T. O'Neill	Hilary Zaloom

The following persons were members of the 1003.2 Balloting Group that approved the standard for submission to the IEEE Standards Board:

Derek Kaufman	*X/Open Institutional Representative*
Shane McCarron	*UNIX International Institutional Representative*
Peter Collinson	*USENIX Association Institutional Representative*

Scott Anderson	Carol J. Harkness	Jim R. Oldroyd
Helene Armitage	Craig Harmer	Craig Partridge
David Athersych	Dale Harris	Rob Peglar
Geoff Baldwin	Myron Hecht	John C. Penney
Jerome E. Banasik	Morris J. Herbert	Rand S. Phares
Steven E. Barber	David F. Hinnant	P. J. Plauger
Robert M. Barned	Lee A. Hollaar	Gerald Powell
David R. Bernstein	Ronald Holt Jr.	Scott E. Preece
Kabekode V. S. Bhat	Randall Howard	James M. Purtilo
Robert Bismuth	Jim Isaak	J. S. Quarterman
Jim Blondran	Richard James	Wendy Rauch-Hindin
Robert Borochoff	Hal Jespersen	Brad Rhoades
Keith Bostic	Greg Jones	Christopher J. Riddick
James P. Bound	Michael J. Karels	Andrew K. Roach
Joseph Boykin	Lorraine C. Kevra	Arnold Robbins
Kevin Brady	Alan W. Kiecker	R. Hughes Rowlands
Phyllis Eve Bregman	Jeff Kimmel	Robert Sarr

A. Winsor Brown
F. Lee Brown Jr.
Luis-Felipe Cabrera
Nicholas A. Camillone
Andres Caravallo
Steven L. Carter
John Caywood
Kilnam Chon
Chan F. Chong
Robert L. Claeson
Mark Colburn
Kenneth N. Cole
Richard Cornelius
William M. Corwin
Mike R. Cossey
William Cox
Donald W. Cragun
Terence Dowling
Stephen A. Dum
John D. Earls
Ron Elliott
Richard W. Elwood
David Emery
Philip H. Enslow
Ken Faubel
Terence Fong
Ed Frankenberry
John A. Gertwagen
Al Gettier
Michel Gien
Gregory W. Goddard
Robert C. Groman
Judy Guist
Gregory Guthrie
Michael J. Hannah
M. J. Kirk
Kenneth C. Klingman
Joshua W. Knight
David Korn
Takahiko Kuki
Robin B. Lake
Mike Lambert
Doris Lebovits
Maggie Lee
Greger Leijonhufvud
Robert M. Lenk
David Lennert
Mark E. Levine
Kevin Lewis
Kin F. Li
James P. Lonjers
Joseph F. P. Luhukay
Paul Lustgarten
Ron Mabe
Robert J. Makowski
Roger J. Martin
Joberto S. B. Martins
Yoshihiro Matsumoto
Shane McCarron
Martin J. McGowan III
Marshall Kirk McKusick
Robert W. McWhirter
Doug Michels
Gary W. Miller
James M. Moe
J. W. Moore
Anita Mundkur
Martha Nalebuff
Fred Noz
Alan F. Nugent
Norman Schneidewind
Wolfgang Schwabl
Richard Scott
Glen Seeds
Dan Shia
Roger Shimada
Mukesh Singhal
Richard Sniderman
Steven Sommars
Bryan W. Sparks
Richard Stallman
Daniel Steinberg
Douglas H. Steves
Peter Sugar
Scott A. Sutter
Ravi Tavakley
Donn Terry
Gary F. Tom
A. T. Twigger
Mark-Rene Uchida
L. David Umbaugh
Michael W. Vannier
M. B. Wagner
John W. Walz
Alan G. Weaver
Larry Wehr
Bruce Weiner
Brian Weis
Peter J. Weyman
Andrew E. Wheeler
David Willcox
Jeff Wubik
Oren Yuen
Jason Zions

When the IEEE Standards Board approved this standard on *<date to be provided>*, it had the following membership:

(to be pasted in by IEEE)

Information technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities

Section 1: General

1.1 Scope

This standard defines a standard source code level interface to command interpretation, or ‘‘shell,’’ services and common utility programs for application programs. These services and programs are complementary to those specified by ISO/IEC 9945-1: 1990 {8}, hereinafter referred to as ‘‘POSIX.1 {8}.’’

The standard has been designed to be used by both application programmers and system implementors. However, it is intended to be a reference document and not a tutorial on the use of the services, the utilities, or the interrelationships between the utilities.

The emphasis of this standard is on the shell and utility functionality required by application programs (including ‘‘shell scripts’’) and not on the direct interactive use of the shell command language or the utilities by humans.

Portions of this standard comprise optional language bindings to system service interfaces. See, for example, the C Language Bindings Option in Annex B. This standard is intended to describe language interfaces and utilities in sufficient detail so that an application developer can understand the required interfaces without access to the source code of existing implementations on which they may be based. Therefore, it does not attempt to describe the source programming language or internal design of the utilities; they should be considered ‘‘black boxes’’ that exhibit the described functionality.

For language interfaces, or functions, this standard has been defined exclusively at the source code level. The objective is that a conforming portable application source program can be translated to execute on a conforming implementation. The standard assumes that the source program may need to be retranslated to produce target code for a new environment prior to execution in that environment.

There is no requirement that the base operating system supporting the shell and utilities be one that fully conforms to ISO/IEC 9945-1: 1990 {8}. (The base system could contain a subset of POSIX.1 {8} functionality, enough to support the requirements for this standard, as described in 2.9.1, but that could not claim full conformance to all of POSIX.1 {8}.) Furthermore, there is no requirement that the shell command interpreter or any of the standard utilities be written as POSIX.1 {8} conforming programs, or be written in any particular language.

Although not requiring a fully conforming POSIX.1 {8} base, this standard is based upon documentation and the knowledge of existing programs that assume an interface and architecture similar to that described by POSIX.1 {8}. Any questions regarding the definition of terms or the semantics of an underlying concept should be referred to POSIX.1 {8}.

1.1.1 Scope Rationale. *(This subclause is not a part of P1003.2)*

This standard is one of a family of related standards. The term POSIX is correctly used to describe this family, and not only its foundation, the operating system interfaces of POSIX.1 {8}. Therefore, POSIX.2 could colloquially be described as the ‘POSIX Shell and Tools Standard.’’

The interfaces documented for this standard are to and from high-level language application programs and to and from the utilities themselves; the standard does not directly address the interface with users.

The ‘‘source code’’ interface to the command interpreter is defined in terms of high-level language functions in 7.1.1 or 7.1.2 (such as *system*(), B.3.1, or *popen*(), B.3.2). There are also other function interfaces, such as those for matching regular expressions in 7.3 (*regcomp*() in B.5). Many of the utilities in this standard, and the shell itself, also accept their own command languages or complex directives as input data, which is also referred to as source code. This data, an ordered series of characters, may be stored in files, or ‘‘scripts,’’ that are portable between systems without true recompilation. However, just as with POSIX.1 {8}, the standard addresses only the issue of source code portability between systems; applications using these calls may have to be recompiled or translated when moving from one system to another.

There has been considerable debate concerning the appropriate scope of the work represented by this standard. The following are rational alternatives that have been evaluated:

(1) Define the shell and tools as extensions to POSIX.1 {8}. This would require a full conforming POSIX.1 {8} system as a base for the new facilities described here. Vocal proponents for this view have been the members of the POSIX.3 working group, who foresaw difficulties in

producing a verification suite standard without having a known operating system base.

(2) Decouple the shell and tools entirely from POSIX.1 {8}. This would potentially allow the standard to be implemented on such popular operating systems as MVS/TSO, VM/CMS, MS/DOS, VMS, etc. Those systems would not have to provide every minor detail of the POSIX.1 {8} language interfaces to conform under this model—only enough to support the shell and tools.

(3) Compromise between options 1 and 2. Base the standard on an interface *similar* to POSIX.1 {8}, but don't require full conformance. A simple example would be a Version 7 UNIX System, which could not conform to POSIX.1 {8} without considerable modification. However, a vendor could support all of the features of this standard without changing its kernel or binary compatibility. Another example would be a system that conformed to all stated POSIX.1 {8} interfaces, but that didn't have a fully conforming C Standard {7} compiler. The difficulty with this option is that it makes the stated goal of the working group a bit fuzzier and increases the amount of analysis required for the features included.

The working group selected option 3 as its goal. It chose to retain the full UNIX system-like orientation, but did not wish to arbitrarily deprive legitimate systems that could *almost* conform. No useful feature of shells or commonly-used utilities were discarded to accommodate nonconforming base systems; on the other hand, no deliberate obstacles were arbitrarily erected. Furthermore, POSIX.1 {8} is still required for its definitions and architectural concepts, which are purposely not repeated in this standard.

One concrete example of how the two standards interrelate is in the usage of POSIX.1 {8} function names in the descriptions of utilities in POSIX.2. There are a number of historical commands that directly mapped into one of the UNIX system calls. For example: `chmod` and *chmod*(); `ln` and *link*(). The POSIX.2 working group was faced with the problem of having to define all of the complex interactions ''behind the scenes'' for some simple commands. Creating a file, for example, involves many POSIX.1 {8} concepts, including processes, user IDs, multiple group permissions (which are optional), error conditions, etc. Rather than enumerating all of these interactions in many places, the POSIX.2 group chose to employ the POSIX.1 {8} function descriptions, where appropriate. See the `chmod` utility in 4.7 as an example. The utility description includes the phrase:

> . . . performing actions equivalent to the *chmod*() function as defined in the POSIX.1 {8} *chmod*() function:

This means that the POSIX.2 implementor has to read the POSIX.1 {8} *chmod*() description and fully understand all of its functionality, requirements, and side effects, which now don't have to be repeated here. (Admittedly, this makes the POSIX.2 standard a bit more difficult to read, but the working group felt that precision transcended the need for readable or semi-tutorial documents.)

The Introduction states that one of the goals of the working group was: ''This interface should be implementable on conforming POSIX.1 {8} systems.'' This implies that the working group has attempted to ensure that no additional functionality or extension is required to implement this standard on the base defined by POSIX.1 {8}. This is not to say that extensions are not allowed, but that they should not be necessary. The goal ‘‘(7) Utilities and standards for the installation of applications" was once interpreted to mean that an elaborate series of tools was required to install and remove applications, based on complex description files and system databases of capabilities. An attempt to provide this was rejected by the balloting group and that type of system is now being evaluated by the POSIX.7 System Administration group. However, the original goal remains in the list, because many of the standard utilities are, in fact, targeted specifically for application installation— `make`, `c89`, `lex`, etc.

1.1.1.1 Existing Practice. *(This subclause is not a part of P1003.2)*

The working group would have been very happy to develop a standard that allowed all historical implementations (i.e., those existing prior to the time of publication) to be fully conforming and all historical applications to be Strictly Conforming POSIX Shell Applications without requiring any changes. Some modifications will be required to reconcile the specific differences between historical implementations; there are many divergent versions of UNIX systems extant and applications have sometimes been written to take advantage of features (or bugs) on specific systems. Therefore, the working group established a set of goals to maximize the value of the standard it eventually produced. These goals are enumerated in the following subclauses. They are listed in approximate priority sequence, where the first subclause is the most important portability goal.

1.1.1.1.1 Preserve Historical Applications

The most important priority was to ensure that historical applications continued to operate on conforming implementations. This required the selection of many utilities and features from the most prevalent historical implementations. The working group is relying on the following factors:

(1) Many inconsistent historical features will still be supported as *obsolescent*.

(2) Common features of System V and BSD will continue to be supported by their sponsors, even if they aren’t included here (just as long as they are not prevented from existing).

Therefore, the standard was written so that the large majority of well-written historical applications should continue to operate as Conforming POSIX Shell Applications Using Extensions.

1.1.1.1.2 Clean Up the Interfaces

The working group chose to extend the benefits of historical UNIX systems by making limited improvements to the utility interfaces; numerous complaints have been heard over the years about the inconsistencies in the command line interface, which have allegedly made it harder for novice users. Given the constraints of **Preserve Historical Applications**, the working group has made the following general modifications:

(1) Utilities have been extended to deal with differences in character sets, collating sequences, and some cultural aspects relating to the locale of the user. (Examples: new features in regular expressions; new formatting options in `date`; see 4.15.)

(2) The utility syntax guidelines in 2.10.2 have been applied to almost all of the utilities to promote a consistent interface. The guidelines themselves have been loosened up a bit from their counterparts in the *SVID*. In many cases historical utilities have not conformed with these guidelines (which were written considerably later than the utilities themselves). The older interfaces have been maintained in the standard as obsolescent features. (Examples: `join`, `sort`.) However, in some cases, such as `dd` and `find`, such major surgery was required that the working group decided to leave the historical interfaces as is. ''Fixing'' the interface would mean replacing the command, which would not help applications portability. So, fixing was limited to relatively minor abuses of the new guidelines, where reasonable consistency could be achieved while still maintaining the general type of interface of the historical version.

(3) Features that were not generally portable across machine architectures or systems have been removed or marked obsolescent and new, more portable interfaces have been introduced. (Examples: the octal number methods of describing file modes in `chmod` and other utilities have been marked obsolescent; the symbolic '`ugo`' method has been extended to other utilities, such as `umask`.)

(4) Features that have proved to be popular in some specific UNIX system variants have been adopted. (Examples: `diff −c`, which originated in BSD systems, and the ''new'' `awk`, from System V.) Such features were selected given the requirements for balloting group consensus; the features had to be used widely enough to balance accusations of ''creeping featurism'' and violations of the UNIX system ''tools philosophy.''

(5) Unreasonable inconsistencies between otherwise similar interfaces have been reconciled. (Example: methods of specifying the patterns to the three `grep`-related utilities have been made more consistent in the standard's single `grep`.)

(6) When irreconcilable differences arose between versions of historical utilities, new interfaces (utility names or syntax) were sometimes added in their places. The working group resisted the urge to deviate significantly from historical practice; the new interfaces are generally consistent with the philosophy of historical systems and represent comparable functionality to the interfaces being replaced. In some cases, System V and BSD had diverged (such as with `echo` and `sum`) so significantly that no compromises for a common interface were possible. In these cases, either the divergent features were omitted or an entirely new command name was selected (such as with `printf` and `cksum`).

(7) Arbitrary limits to utility operations have been removed. (Example: some historical `ed` utilities have very limited capabilities for dealing with large files or long input lines.)

(8) Arbitrary limitations on historical extensions have been eliminated. (Example: regular expressions have been described so that the popular `\<` . . . `\>` extension is allowed.)

(9) Input and output formats have been specified in more detail than historical implementations have required, allowing applications to more effectively operate in pipelines with these utilities. (Example: `comm`.)

Thus, in many cases the working group could be accused of ‘‘violating Existing Practice,’’ and in fact received some balloting objections to that effect from implementors (although rarely from users or application developers). The working group was sensitive to charges that it was engaged in arbitrary software engineering rather than merely codifying existing practice. When changes were made, they were always written to preserve historical applications, but to move new conforming applications into a more consistent, portable environment. This strategy obviously requires changes to historical implementations; the working group carefully evaluated each change, weighing the value to users against the one-time costs of adding the new interfaces (and of possibly breaking applications that took advantage of bugs), generally siding with the users when the costs to implementations and applications was not excessively high.

In some cases, changes were reluctantly made that could conceivably break some historical applications; the working group allowed these only in the face of practices it considered rare or significantly misguided.

1.1.1.1.3 Allow Historical Conforming Applications

It is likely that many historical shell scripts will be Strictly Conforming POSIX.2 Applications without requiring modifications. Developers have long been aware of the differences among the historical UNIX system variants and have avoided the nonportable aspects to increase the scope of their applications’ marketplace. However, the previous goal of a consistent interface was considered to be quite important, so there will be modifications required to some applications if they wish to be maximally portable in the future.

1.1.1.1.4 Preserve Historical Implementations

As explained in 1.1.1.1.2, the requirements for portability and a consistent interface have caused the working group to add new utilities and features. No historical implementations contained all of the attributes required by the working group. Therefore, this lowest priority goal fell victim to the preceding goals, and every known historical implementation will require some modifications to conform to this standard.

The working group took care to ensure that the implementations could add the new or modified features without breaking the operation of existing applications. (Note that the standard utilities are not considered applications in this regard, but are part of the implementation. In fact, many or most of the utilities named by this standard will have to change to some extent.)

1.1.1.2 Outside the Scope. *(This subclause is not a part of P1003.2)*

The following areas are outside the scope of this standard. This subclause explains more of the rationale behind the exclusions. (It should be noted that this is not an official list. It was not part of the Project Authorization Request submitted to the IEEE , but was devised as a guide to keep the working group discussions on track.)

(1) *Operating system administrative commands (privileged processes, system processes, daemons, etc.).*

The working group followed the lead of the POSIX.1 {8} group in this instance. Administrative commands were felt to be too implementation dependent and not useful for application portability. Subsequent to this decision, a separate POSIX.7 working group was formed to deal with this area of ''operator portability.'' It is anticipated that utilities needed for system administration will be closely coordinated with the POSIX.2 working group.

(2) *Commands required for the installation, configuration, or maintenance of operating systems or file systems.*

This area is similar to item (1). System installation is contrasted against the application installation portion of the Scope by its orientation to installing the operating system itself, versus application programs. The exclusion of operating system installation facilities should not be interpreted to mean that the application installation procedures *cannot* be used for installing operating system components. The proposed interface for this area encountered stiff resistance from the balloting group in Draft 8 and was temporarily withdrawn. As described in Annex E.4, a decision of the balloting group is pending on whether to begin work on a supplement to this standard (POSIX.2b) for application installation.

(3) *Networking commands.*

These were excluded because they are deeply involved with other standards making bodies and are probably too complicated. In this case, several working groups were formed within the POSIX family to deal with this. It is anticipated that utilities needed for networking, if any, will be closely coordinated with the POSIX.2 working group. (In early drafts of this standard, which predated the formation of the networking-specific POSIX working groups, the historical 'UNIX system to UNIX system copy [UUCP]'' programs and protocols were included. These descriptions have been removed in deference to a more appropriate working group.)

(4) *Terminal control or user-interface programs (e.g., visual shells, visual editors, window managers, command history mechanisms, etc.).*

This is probably the most contentious exclusion. A common complaint about many UNIX systems is how they're not very ‘‘user friendly.’’ Some people have hoped that the interface to users could be standardized with mice, icon-based desktop metaphors, and so forth. This standard neatly sidesteps those concerns by reminding its audience that it is an application portability standard, and therefore has little relationship to the manner in which users manage their terminals.

However, this guideline was not meant to apply to applications. It is perfectly reasonable for an application to assume it can have a user interacting with it. That is why such facilities as displaying strings (with `printf`) without `<newline>`s, `stty`, and various prompting utilities are included in the standard.

The interfaces in this standard are very oriented to command lines being issued by shell scripts, or through the *system* () or *popen* () functions. Therefore, interactive text editors, pagers, and other user interface tools have been omitted for now. Alternatively, other standards bodies, such as X3H3.6 and the IEEE TCOS P1201 working group, are devising interfaces that could possibly be more useful and long-lived than any prescribed by POSIX. 2.

There is one area of this subject that will be addressed by POSIX. 2. The scope of the working group has been expanded to include what is being termed the *User Portability Extension* , POSIX. 2a. This will be published as a supplement to this standard and have the goal of providing a portable environment for relatively expert time-sharing or software development users. It will not attempt to deal with mice or windows or other advanced interfaces at this time, but should cover many of the terminal-oriented utilities, such as a full-screen editor, currently avoided by this edition of POSIX. 2.

(5) *Graphics programs or interfaces.*

See the comments on user interface, above.

(6) *Text formatting programs or languages.*

The existing text formatting languages are generally too primitive in scope to satisfy many users, who have relied on a myriad of macro languages. There is an ISO standard text description language, SGML , but this has had insufficient exposure to the UNIX system community for standardization as part of POSIX at this time.

(7) *Database programs or interfaces (e.g. SQL , etc.).*

These interfaces are the province of other standards bodies.

1.1.1.3 Language-Independent Descriptions. *(This subclause is not a part of P1003.2)*

The POSIX.1 {8} and POSIX.5 working groups are currently engaged in developing the model for language-independent descriptions of system services. When complete, it will allow the C language bias of the POSIX.1 {8} standard to be excised and C will take its place among other language bindings that interface with the core services descriptions. The POSIX.2 working group did not wish to duplicate effort, and has therefore waited until POSIX.1 {8} achieves progress in this area. Thus, like the first version of POSIX.1 {8}, the initial drafts of POSIX.2 start life as a C-only standard, with language independence scheduled to be included in a later draft. Fortunately, this standard is substantially less involved with C than POSIX.1 {8} is. In fact, all of the C interfaces are entirely optional.

1.1.1.4 Base Documents. *(This subclause is not a part of P1003.2)*

The working group consulted a number of documents in the course of its deliberations, to select utilities and features. There were five primary documents that started off the process:

(1) The *System V Interface Definition* (*SVID*), Issue 2, Volume 2.

(2) The *X/Open Portability Guide*, (*XPG*), Issues II and III, Volume 1.

(3) *The UNIX User's Reference Manual*, 4.3 Berkeley Software Distribution, Virtual VAX-11 Version. (The printed documentation as well as the online versions provided with the BSD ''Tahoe'' and ''Reno'' distributions were considered as one base document for the POSIX.2 work.)

(4) *The KornShell Command and Programming Language*, by Bolsky and Korn.

(5) *The AWK Programming Language*, by Aho, Kernighan, and Weinberger.

The *XPG* was used most heavily in initial deliberations about which utilities and features to include. The X/Open companies had done a very thorough job in analyzing the *SVID* and other standards to compile a list of the most useful and portable utilities. They carefully marked many features that had portability problems and the working group avoided them for this standard.

AT&T, X/Open, and Berkeley provided machine-readable documentation for the use of the working group. However, due to very substantial differences in formatting standards, there is little resemblance between some of the utilities described here and their cousins in the *SVID*, *XPG*, and BSD user manual. Nevertheless, early usage of these documents was an invaluable aid in the production of the standard and the POSIX.2 working group extends its sincere thanks to all three organizations for their generous cooperation.

The biggest divergence in POSIX.2's documentation has been its philosophy of fully specifying interfaces. The *SVID* and *XPG* are oriented solely towards application portability. Implementors would have a difficult time writing some of these utilities from the descriptions alone. In fact, both documents freely rely on the potential implementors licensing the source code for the reference systems to complete the specification. The POSIX.2 standard, on the other hand, also has implementors in its audience and it strove to expand its descriptions wherever useful and feasible. For example, it makes use of BNF grammars to describe complex syntaxes. It attempts to describe the interactions between options, operands, and environment variables, where conflicts can exist. It also attempts to describe all of the useful utility input and output formats. The goal here was to allow application developers to write filters or other programs that could parse the output of any of these utilities or to provide meaningful input from their programs. To the working group's knowledge, this is a task never before attempted for the historical UNIX system commands—the source code was always so readily available to anyone who really needed to know this information.

The two commercial books listed were used as reference materials in preparing information on the shell and the *awk* language that was more recent and complete than AT&T's or X/Open's documentation.

1.1.1.5 History. *(This subclause is not a part of P1003.2)*

The *1984 /usr/group Standard* was originally intended to include the shell and user level commands. However, the /usr/group (now known as ‘‘UniForum’’) Standards Committee was unable to begin this effort, due to the complexity of the system call and library functions that it eventually did publish.

A shell was referred to in the *system*() function defined by *ANSI /X3.159-1989 Programming Language C Standard*, but no syntax for the shell command language was attempted.

As the first version of POSIX.1 {8} neared completion, it became apparent that the usefulness of POSIX would be diminished if no shell or utilities were defined. Therefore, the POSIX.2 working group was formed in January 1986 at the Denver, Colorado, meeting of POSIX.1 {8} to address this concern.

The progress of the working group has seemed rather slow during the more than three years of its existence. This is primarily because its membership had substantial overlap with the POSIX.1 {8} working group; for example, the Chair of POSIX.2 was also the Technical Editor of POSIX.1 {8} (and POSIX.2 as well!) at the time. And, meetings were arbitrarily shortened to allow the POSIX.1 {8} group to move forward as quickly as possible.

1.1.1.6 Internationalization. *(This subclause is not a part of P1003.2)*

Some of the utilities and concepts described in this standard contain requirements that standardize multilingual and multicultural support. Most of the internationalized support for this standard was proposed by the UniForum Technical Committee Subcommittee on Internationalization, at the request of the POSIX.2 working group.

UniForum, a nonprofit organization, organizes subcommittees of Technical Committees to do standards research on different topics pertinent to POSIX. The

UniForum Subcommittee on Internationalization is one such group. It was formed to propose and promote standard internationalized extensions to POSIX-based systems. The POSIX.2 working group and the UniForum Subcommittee on Internationalization coordinated their work by the use of liaison members, who attended the meetings of both groups. The interaction between the two groups started when POSIX.2 asked the Subcommittee on Internationalization to provide internationalized support for regular expressions. Later, the Subcommittee on Internationalization was charged with identifying areas in the standard needing changes for internationalized support and proposing those changes.

1.1.1.7 Test Methods. *(This subclause is not a part of P1003.2)*

The POSIX.3 working group has worked on a test methods specification for verifying conformance to POSIX standards in general and POSIX.1 {8} and POSIX.2 in particular. Test methods for POSIX.2 should be published as a separate document [1] sometime after POSIX.2 is approved.

1.1.1.8 Organization of the Standard. *(This subclause is not a part of P1003.2)*

The standard document is organized into sections. Some of these, such as the Scope in 1.1, are mandated by ISO/IEC, the IEEE, and other standards bodies. The remainder of the document is organized into small sections for the convenience of the working group and others. It has been suggested that all of the utility descriptions (and maybe the functions, too) should be lumped into one large section, all in alphabetical order. This would presumably make it easier for some users to use the document as a reference document. The working group deliberately chose to not organize it in this way, for the following reasons:

(1) Certain sections are optional. It is more convenient for the document's internal references, and also for people specifying systems, if these optional sections are in large pieces, rather than a detailed list of utility names.

(2) Future supplements to this standard will be adding new utilities that will also be optional. It would be confusing to try to merge documents at a level below major sections (chapters).

1) See the Foreword for information on the activities of other POSIX working groups.

1.2 Normative References

The following standards contain provisions which, through references in this text, constitute provisions of this standard. At the time of publication, the editions indicated were valid. All standards are subject to revision, and parties to agreements based on this part of this International Standard are encouraged to investigate the possibility of applying the most recent editions of the standards listed below. Members of IEC and ISO maintain registers of currently valid International Standards.

{1} ISO/IEC 646: 1983,[2] *Information processing— ISO 7-bit coded character set for information interchange.*

{2} ISO 1539: 1980, *Programming languages— FORTRAN.*

{3} ISO 4217: 1987, *Codes for the representation of currencies and funds.*

{4} ISO 4873: 1986, *Information processing— ISO 8-bit code for information interchange—Structure and rule for implementation.*

{5} ISO 8859-1: 1987, *Information processing—8-bit single-byte coded graphic character sets—Part 1: Latin alphabet No. 1.*

{6} ISO 8859-2: 1987, *Information processing—8-bit single-byte coded graphic character sets—Part 2: Latin alphabet No. 2.*

{7} ISO/IEC 9899: 1990, *Information processing systems—Programming* 1
languages—C.

{8} ISO/IEC 9945-1: 1990, *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]*

1.3 Conformance

1.3.1 Implementation Conformance

1.3.1.1 Requirements

A *conforming implementation* shall meet all of the following criteria:

---

2) Under revision. (This notation is meant to explicitly reference the 1990 Draft International Standard version of ISO/IEC 646.)

ISO/IEC documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.

(1) The system shall support all required interfaces defined within this standard. These interfaces shall support the functional behavior described herein. The system shall provide the shell command language described in Section 3 and the utilities in Section 4.

(2) The system may provide one or more of the following: the Software Development Utilities Option, the C Language Bindings Option, the C Language Development Utilities Option, the FORTRAN Development Utilities Option, or the FORTRAN Runtime Utilities Option. When an implementation claims that an optional facility is provided, all of its constituent parts shall be provided.

(3) The system may provide additional or enhanced utilities, functions, or facilities not required by this standard. Nonstandard extensions should be identified as such in the system documentation. Nonstandard extensions, when used, may change the behavior of utilities, functions, or facilities defined by this standard. In such cases, the implementation's conformance document (see 2.2.1.2) shall define an execution environment (i.e., shall provide general operating instructions) in which an application can be run with the behavior specified by the standard. In no case shall such an environment require modification of a Strictly Conforming POSIX.2 Application.

1.3.1.2 Documentation

A conformance document with the following information shall be available for an implementation claiming conformance to this standard. The conformance document shall have the same structure as this standard, with the information presented in the appropriately numbered sections; sections that consist solely of subordinate section titles, with no other information, are not required.

The conformance document shall not contain information about extended facilities or capabilities outside the scope of this standard, unless those extensions affect the behavior of a Strictly Conforming POSIX.2 Application; in such cases, the documentation required by the previous subclause shall be included.

The conformance document shall contain a statement that indicates the full name, number, and date of the standard that applies. The conformance document may also list software standards approved by ISO/IEC or any ISO/IEC member body that are available for use by a Conforming POSIX.2 Application. It should indicate whether it is based on a fully-conformant POSIX.1 {8} system. Applicable characteristics where documentation is required by one of these standards, or by standards of government bodies, may also be included.

The conformance document shall describe the symbolic values found in 2.13.2, stating values, the conditions under which those values can change, and the limits of such variations, if any.

The conformance document shall describe the behavior of the implementation for all implementation-defined features defined in this standard. This requirement shall be met by listing these features and providing either a specific reference to the system documentation or providing full syntax and semantics of these features. When the value or behavior in the implementation is designed to be variable or customizable on each instantiation of the system, the implementation provider shall document the nature and permissible ranges of this variation. When information required by this standard is related to the underlying operating system and is already available in the POSIX.1 {8} conformance document, the implementation need not duplicate this information in the POSIX.2 conformance document, but may provide a cross-reference for this purpose.

The conformance document may specify the behavior of the implementation for those features where this standard states that implementations may vary or where features are identified as undefined or unspecified.

No specifications other than those described in this subclause (1.3.1.2) shall be present in the conformance document.

The phrase ‘‘shall be documented’’ in this standard means that documentation of the feature shall appear in the conformance document, as described previously, unless the system documentation is explicitly mentioned.

The system documentation should also contain the information found in the conformance document.

1.3.1.3 Conforming Implementation Options

The following symbolic constants, described in 2.13.2 reflect implementation options for this standard that could warrant requirement by Conforming POSIX.2 Applications, or in specifications of conforming systems, or both:

{POSIX2_SW_DEV}	The system supports the Software Development Utilities Option in Section 6.
{POSIX2_C_BIND}	The system supports the C Language Bindings Option in Annex B.
{POSIX2_C_DEV}	The system supports the C Language Development Utilities Option in Annex A.
{POSIX2_FORT_DEV}	The system supports the FORTRAN Development Utilities Option in Annex C.
{POSIX2_FORT_RUN}	The system supports the FORTRAN Runtime Utilities Option in Annex C.
{POSIX2_LOCALEDEF}	The system supports the creation of locales as described in 4.35.

Additional language bindings and development utility options may be provided in other related standards or in future revisions to this standard. In the former case, additional symbolic constants of the same general form as shown in this subclause should be defined by the related standard document and made available to the application, without requiring this POSIX.2 document to be updated.

1.3.2 Application Conformance

All applications claiming conformance to this standard fall within one of the following categories:

1.3.2.1 Strictly Conforming POSIX.2 Application

A Strictly Conforming POSIX.2 Application is an application that requires only the facilities described in this standard (including any required facilities of the underlying operating system; see 2.9.1). Such an application:

(1) shall accept any implementation behavior that results from actions it takes in areas described in this standard as *implementation-defined* or *unspecified*, or where the standard indicates that implementations may vary;

(2) shall not perform any actions that are described as producing *undefined* results;

(3) for symbolic constants, shall accept any value in the range permitted by this standard, but shall not rely on any value in the range being greater than the minimums listed in this standard;

(4) shall not use facilities designated as *obsolescent*;

(5) is required to tolerate, and is permitted to adapt to, the presence or 1
absence of optional facilities whose availability is indicated by the con- 1
stants in 2.13.1, or that are described using the verb *may*. However, an 1
application requiring a high-level language binding option can only be considered at best a Conforming POSIX.2 Application; see 1.3.2.2.

Within this standard, any restrictions placed upon a Conforming POSIX.2 Application shall also restrict a Strictly Conforming POSIX.2 Application.

1.3.2.2 Conforming POSIX.2 Application

The term Conforming POSIX.2 Application is used to describe either of the two following application types.

1.3.2.2.1 ISO/IEC Conforming POSIX.2 Application

An ISO/IEC Conforming POSIX.2 Application is an application that uses only the facilities described in this standard (including the implied facilities of the underlying operating system; see 2.9.1) and approved conforming language bindings for any ISO/IEC standard. Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other ISO/IEC standards used.

1.3.2.2.2 <National Body> Conforming POSIX.2 Application

A <National Body> Conforming POSIX.2 Application differs from an ISO/IEC Conforming POSIX.2 Application in that it also may use specific standards of a single ISO/IEC member body referred to here as ‘‘*<National Body>*.’’ Such an application shall include a statement of conformance that documents all options and limit dependencies, and all other *<National Body>* standards used.

1.3.2.3 Conforming POSIX.2 Application Using Extensions

A Conforming POSIX.2 Application Using Extensions is an application that differs from a Conforming POSIX.2 Application only in that it uses nonstandard facilities that are consistent with this standard. Such an application shall fully document its requirements for these extended facilities, in addition to the documentation required of a Conforming POSIX.2 Application. A Conforming POSIX.2 Application Using Extensions shall be either an ISO/IEC Conforming POSIX.2 Application Using Extensions or a <National Body> Conforming POSIX.2 Application Using Extensions (see 1.3.2.2.1 and 1.3.2.2.2).

1.3.3 Conformance Rationale. *(This subclause is not a part of P1003.2)*

These conformance definitions are closely related to those in POSIX.1 {8}.

The terms *Conforming POSIX.2 Application* and its variants were selected to parallel the terms used in POSIX.1 {8}.

The descriptions of the ISO/IEC and <National Body> Conforming POSIX.2 Applications are similar to the same descriptions in POSIX.1 {8}. This is not a duplication of effort, as this standard relies on only a portion of POSIX.1 {8}, as explained in 1.1 and 2.9.1. Therefore conformance to POSIX.2 has to be described separately from any conformance options or requirements in POSIX.1 {8}.

A reference to a Language-Independent System Services Option was removed from the list of optional features that may be provided by the conforming implementation. There is no conformance value provided by that section, except as a reference point for functions actually provided by a real language binding. Therefore, the language binding sections are the ones that remain in the optional list. The Draft 8 section Language-Dependent Services for the C Programming Language was removed, as this subject is adequately, and appropriately, covered in Annex A.

The documentation requirement for implementation extensions (‘‘shall define an execution environment’’) is simply meant to require that system-wide or per-user configuration options or environment variables that affect the operation of applications that use the standard utilities and functions be described in the conformance document. For example, if setting the (imaginary) **LC_TRUTH** variable causes changes in the exit status of `true`, the conformance document must describe this condition and how to avoid it—say, by unsetting the variable in the login script.

For further rationale on the types of conformance, see the POSIX.1 {8} Rationale.

Section 2: Terminology and General Requirements

2.1 Conventions

2.1.1 Editorial Conventions

This standard uses the following editorial and typographical conventions. A summary of typographical conventions is shown in Table 2-1.

Table 2-1 – Typographical Conventions

Reference	Example
C-Language Data Type	*long*
C-Language Function	*system* ()
C-Language Function Argument	*arg1*
C-Language Global External	*errno*
C-Language Header	<sys/stat.h>
C-Language Keyword	#define
Cross Reference: Annex	Annex A
Cross Reference: Clause	2.3
Cross Reference: Other Standard	ISO 9999-1 {*n*}
Cross Reference: Section	Section 2
Cross Reference: Subclause	2.3.4, 2.3.4.5, 2.3.4.5.6
Defined Term	(see text)
Environment Variable	**PATH**
Error Number	[EINTR]
Example Input	echo foo
Example Output	**foo**
Figure Reference	Figure 7
File Name	/tmp
Parameter	<*directory pathname*>
Special Character	<newline>
Symbolic Constant, Limit	{_POSIX_VDISABLE}, {LINE_MAX}
Table Reference	Table 6
Utility Name	awk
Utility Operand	*file_name*
Utility Option	−c
Utility Option with Option-Argument	−w *width*

The **Bold Courier** font is used to show brackets that denote optional arguments in a utility synopsis, as in

cut **[**−c *list* **]** **[***file_name* **]**

These brackets shall not be used by the application unless they are specifically mentioned as literal input characters by the utility description.

There are two types of symbols enclosed in angle brackets (< >):

C-Language Headers The header name is in the Courier font, such as <sys/stat.h>. When coding C programs, the brackets are used as required by the language.

Parameters Parameters, also called *metavariables*, are in italics, such as <*directory pathname*>. The entire symbol, including the brackets, is meant to be replaced by the value of the symbol described within the brackets.

Numbers within braces, such as ''POSIX.1 {8},'' represent cross references to the Normative References clause (see 1.2). If the number is preceded by a B, it represents a Bibliographic entry (see Annex D). Bibliographic entries are for information only.

In some examples, the **Bold Courier** font is used to indicate the system's output that resulted from some user input, shown in Courier.

Defined terms are shown in three styles, depending on context:

(1) Terms defined in 2.2.1, 2.2.2, and 3.1 are expressed as subclause titles. Alternative forms of the terms appear in [brackets].

(2) The initial appearances of other terms, applying to a limited portion of the text, are in *italics*.

(3) Subsequent appearances of the term are in the Roman font.

Symbolic constants are shown in two styles: those within curly braces are intended to call the reader's attention to values in <limits.h> and <unistd.h>; those without braces are usually defined by one or a few related functions. There is no semantic difference between these two forms of presentation.

Filenames and pathnames are shown in Courier. When a pathname is shown starting with ''$HOME/'', this indicates the remaining components of the pathname are to be related to the directory named by the user's **HOME** environment variable.

The style selected for some of the special characters, such as <newline>, matches the form of the input given to the localedef utility (see 2.5.2). Generally, the characters selected for this special treatment are those that are not visually distinct, such as the control characters <tab> or <newline>.

Literal characters and strings used as input or output are shown in various ways, depending on context:

`%`, `begin` When no confusion would result, the character or string is rendered in the Courier font and used directly in the text.

`'c'` In some cases a character is enclosed in single-quote characters, similar to a C-language character constant. Unless otherwise noted, the quotes shall not be used as input or output.

`"string"` In some cases, a string is enclosed in double-quote characters, similar to a C-language string constant. Unless otherwise noted, the quotes shall not be used as input or output.

Defined names that are usually in lowercase, particularly function names, are never used at the beginning of a sentence or anywhere else that regular English usage would require them to be capitalized.

Parenthetical expressions within normative text also contain normative information. The general typographic hierarchy of parenthetical expressions is:

{ [( )] }

The square brackets are most frequently used to enclose a parenthetical expression that contains a function name [such as *waitpid* ()], with its built-in parentheses.

In some cases, tabular information is presented inline; in others it is presented in a separately-labeled Table. This arrangement was employed purely for ease of reference and there is no normative difference between these two cases.

Annexes marked as *normative* are parts of the standard that pose requirements, exactly the same as the numbered Sections, but have been moved to near the end of the document for clarity of exposition. *Informative* Annexes are for information only and pose no requirements. All material preceding page 1 of the document (the ‘‘front matter’’) and the two indexes at the end are also only informative.

NOTES that appear in a smaller point size and are indented have one of two different meanings, depending on their location:

— When they are within the normal text of the document, they are the same as footnotes—informative, posing no requirements on implementations or applications.

— When they are attached to Tables or Figures, they are normative, posing requirements.

Text marked as examples (including the use of ‘‘e.g.’’) is for information only. The exception to this comes in the C-language programs and program fragments used to represent algorithms, as described in 2.1.3.

The typographical conventions listed here are for ease of reading only. Editorial inconsistencies in the use of typography are unintentional and have no normative meaning in this standard.

2.1.2 Grammar Conventions

Portions of this standard 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 (see A.3). 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 uppercase letters and underscores. Examples: `NEWLINE`, `ASSIGN_OP`, `NAME`.
- The identifiers for nonterminals are all lowercase.

2.1.3 Miscellaneous Conventions

This standard frequently uses the C language to express algorithms in terms of programs or program fragments. The following shall be considered in reading this code:

- The programs use the syntax and semantics described by the C Standard {7}.
- The programs are merely examples and do not represent the most efficient, or only, means of coding the interface. Implementations may use other programming languages or algorithms, as long as the results are the same as those achieved by the programs in this standard.
- C-language comments are informative and pose no requirements.

Further conventions are presented in:

- Utility Conventions, 2.10, describing utility and application command-line syntax
- File Format Notation, 2.12, describing the notation used to represent utility input and output

2.1.4 Conventions Rationale. *(This subclause is not a part of P1003.2)*

The C language was chosen for many examples because:

- It eliminates any requirement to document a different pseudocode.
- It is a familiar language to many of the potential readers of POSIX. 2.
- It is the language most widely used for historical implementations of the utilities.

2.2 Definitions

2.2.1 Terminology

For the purposes of this standard, the following definitions apply:

2.2.1.1 can: The word *can* is to be interpreted as describing a permissible optional feature or behavior available to the application; the implementation shall support such features or behaviors as mandatory requirements.

2.2.1.2 conformance document: A document provided by an implementor that contains implementation details as described in 1.3.1.2.

2.2.1.3 implementation: An object providing to applications and users the services defined by this standard. The word *implementation* is to be interpreted to mean that object, after it has been modified in accordance with the manufacturer's instructions to:

- configure it for conformance with this standard;
- select some of the various optional facilities described by this standard, through customization by local system administrators or operators.

An exception to this meaning occurs when discussing conformance documentation or using the term *implementation defined* . See 2.2.1.4 and 1.3.1.2.

2.2.1.4 implementation defined: When a value or behavior is described by this standard as *implementation defined* , the implementation provider shall document the requirements for correct program construction and correct data in the use of that value or behavior. When the value or behavior in the implementation is designed to be variable or customizable on each instantiation of the system, the implementation provider shall document the nature and permissible ranges of this variation. (See 1.3.1.2.)

2.2.1.5 may: The word *may* is to be interpreted as describing an optional feature or behavior of the implementation that is not required by this standard, but there is no prohibition against providing it. A Strictly Conforming POSIX. 2 Application 1 is permitted to use such features, but shall not rely on the implementation's actions in such cases. To avoid ambiguity, the reverse sense of *may* is not expressed as *may not*, but as *need not*.

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

2.2.1.7 shall: In this standard, the word *shall* is to be interpreted as a requirement on the implementation or on Strictly Conforming POSIX.2 Applications, where appropriate.

2.2.1.8 should: With respect to implementations, the word *should* is to be interpreted as an implementation recommendation, but not a requirement. With respect to applications, the word *should* is to be interpreted as recommended programming practice for applications and a requirement for Strictly Conforming POSIX.2 Applications.

2.2.1.9 system documentation: All documentation provided with an implementation, except the conformance document. Electronically distributed documents for an implementation are considered part of the system documentation.

2.2.1.10 undefined: A value or behavior is *undefined* if the standard imposes no portability requirements on applications for erroneous program construction, erroneous data, or use of an indeterminate value. Implementations (or other standards) may specify the result of using that value or causing that behavior. An application using such behaviors is using extensions, as defined in 1.3.2.3.

2.2.1.11 unspecified: A value or behavior is *unspecified* if the standard imposes no portability requirements on applications for a correct program construction or correct data. Implementations (or other standards) may specify the result of using that value or causing that behavior. An application requiring a specific behavior, rather than tolerating any behavior when using that functionality, is using extensions, as defined in 1.3.2.3.

2.2.1.12 Terminology Rationale *(This subclause is not a part of P1003.2)*

Most of these terms were adapted from their POSIX.1 {8} counterparts with little modification.

The reader is referred to the definition of *program* in 2.2.2.119 to understand the expression ''program construction.'' The use of *program* in this standard is differentiated from POSIX.1 {8}'s emphasis only on high level languages by this standard's broader concern with utility and command language interactions. Included in the scope of program construction are:

(1) Shell command language

(2) Command arguments

(3) Regular expressions, of various types

(4) Command input language syntax, such as `awk`, `bc`, `ed`, `lex`, `make`, `sed`, and `yacc`. Some of these are so complex that they rival traditional high level languages.

The usage of *can* and *may* were selected to contrast optional application behavior (can) against optional implementation behavior (may).

The term *supported* was removed from Draft 8; it had originally been copied from the POSIX.1 {8} document, but it later became clear that its requirement for function ''stubs'' for unsupported functions made little sense in this standard. The term *support* therefore reverts to its English-language meaning.

The term *obsolescent* was changed to *deprecated* in some earlier drafts, but it was restored to match POSIX.1 {8}'s use of the term. It means ''do not use this feature in new applications.'' The obsolescence concept is not an ideal solution, but was used as a method of increasing consensus: many more objections would be heard from the user community if some of these historical features were suddenly withdrawn without the grace period obsolescence implies. The phrase ''may be considered for withdrawal in future revisions'' implies that the result of that consideration might in fact keep those features indefinitely if the predominance of applications does not migrate away from them quickly.

2.2.2 General Terms

For the purposes of this standard, the following definitions apply.

2.2.2.1 absolute pathname: See *pathname resolution* in 2.2.2.104.

2.2.2.2 address space: The memory locations that can be referenced by a process. [POSIX.1 {8}]

2.2.2.3 affirmative response: An input string that matches one of the responses acceptable to the LC_MESSAGES category keyword `yesexpr`, matching an extended regular expression in the current locale; see 2.5.

2.2.2.4 `<alert>`: A character that in the output stream shall indicate that a terminal should alert its user via a visual or audible notification. 1

The `<alert>` shall be the character designated by `'\a'` in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the alert function.

2.2.2.5 angle brackets: The characters ‘`<`’’ (*left-angle-bracket*) and ‘`>`’’ (*right-angle-bracket*).

When used in the phrase ‘‘enclosed in angle brackets’’ the symbol ‘`<`’’ shall immediately precede the object to be enclosed, and ‘`>`’’ shall immediately follow it. When describing these characters in 2.4, the names `<less-than-sign>` and `<greater-than-sign>` are used.

2.2.2.6 appropriate privileges: An implementation-defined means of associating privileges with a process with regard to the function calls and function call options defined in POSIX.1 {8} that need special privileges.

There may be zero or more such means. [POSIX.1 {8}]

2.2.2.7 argument: A parameter passed to a utility as the equivalent of a single string in the *argv* array created by one of the POSIX.1 {8} *exec* functions.

See 2.10.1 and 3.9.1.1. An argument is one of the options, option-arguments, or operands following the command name.

2.2.2.8 asterisk: The character ‘`*`’’.

2.2.2.9 background process: A process that is a member of a background process group. [POSIX.1 {8}]

2.2.2.10 background process group: Any process group, other than a foreground process group, that is a member of a session that has established a connection with a controlling terminal. [POSIX.1 {8}]

2.2.2.11 backquote: The character ‘‘`’’, also known as a *grave accent*.

2.2.2.12 backslash: The character ‘‘\’’, also known as a *reverse solidus*.

2.2.2.13 <backspace>: A character that normally causes printing (or displaying) to occur one column position previous to the position about to be printed.

The <backspace> shall be the character designated by '\b' in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the backspace function. The <backspace> character defined here is not necessarily the ERASE special character defined in POSIX.1 {8} 7.1.1.9.

2.2.2.14 basename: The final, or only, filename in a pathname.

2.2.2.15 basic regular expression: A pattern (sequence of characters or symbols) constructed according to the rules defined in 2.8.3.

2.2.2.16 <blank>: One of the characters that belong to the blank character class as defined via the LC_CTYPE category in the current locale.

In the POSIX Locale, a <blank> is either a <tab> or a <space>.

2.2.2.17 blank line: A line consisting solely of zero or more <blank>s terminated by a <newline>.

See also *empty line* (2.2.2.44).

2.2.2.18 block special file: A file that refers to a device.

A block special file is normally distinguished from a character special file by providing access to the device in a manner such that the hardware characteristics of the device are not visible. [POSIX.1 {8}]

2.2.2.19 braces: The characters ‘‘{’’ (*left brace*) and ‘‘}’’ (*right brace*), also known as *curly braces*.

When used in the phrase ‘‘enclosed in (curly) braces’’ the symbol ‘‘{’’ shall immediately precede the object to be enclosed, and ‘‘}’’ shall immediately follow it. When describing these characters in 2.4, the names <left-brace> and <right-brace> are used.

2.2.2.20 brackets: The characters ‘‘`[`’’ (*left-bracket*) and ‘‘`]`’’ (*right-bracket*), also known as *square brackets* .

When used in the phrase ‘‘enclosed in (square) brackets’’ the symbol ‘‘`[`’’ shall immediately precede the object to be enclosed, and ‘‘`]`’’ shall immediately follow it. When describing these characters in 2.4, the names `<left-square-bracket>` and `<right-square-bracket>` are used.

2.2.2.21 built-in utility: A utility implemented within a shell.

The utilities referred to as *special built-ins* have special qualities, described in 3.14. Unless qualified, the term *built-in* includes the special built-in utilities.

The utilities referred to as *regular built-ins* are those named in Table 2-2. As indicated in 2.3, there is no requirement that these utilities be actually built into the shell on the implementation, but that they do have special command-search qualities.

2.2.2.22 byte: An individually addressable unit of data storage that is equal to 1
or larger than an octet, used to store a character or a portion of a character; see 1
2.2.2.24. 1

A byte is composed of a contiguous sequence of bits, the number of which is imple- 1
mentation defined. The least significant bit is called the *low-order* bit; the most significant is called the *high-order* bit. [POSIX.1 {8}]

NOTE: This definition of *byte* is actually from the C Standard {7} because POSIX.1 {8} merely refer-
ences it without copying the text. It has been reworded slightly to clarify its intent without intro- 1
ducing the C Standard {7} terminology ‘‘basic execution character set,’’ which is inapplicable to this 1
standard. It deviates intentionally from the usage of *byte* in some other standards, where it is used 1
as a synonym for *octet* (always eight bits). On a POSIX.1 {8} system, a byte may be larger than eight 1
bits so that it can be an integral portion of larger data objects that are not evenly divisible by eight 1
bits (such as a 36-bit word that contains 4 9-bit bytes). 1

2.2.2.23 `<carriage-return>`: A character that in the output stream shall indi- 1
cate that printing should start at the beginning of the same physical line in which the `<carriage-return>` occurred.

The `<carriage-return>` shall be the character designated by `'\r'` in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the beginning of the line.

2.2.2.24 character: A sequence of one or more bytes representing a single graphic symbol.

NOTE: This term corresponds in the C Standard {7} to the term *multibyte character* , noting that a single-byte character is a special case of multibyte character. Unlike the usage in the C Standard {7}, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

[POSIX.1 {8}]

(See 2.4 for a further explanation of the graphical representations of characters, or ‘‘glyphs,’’ versus character encodings.)

2.2.2.25 character class: A named set of characters sharing an attribute associated with the name of the class.

The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale; see 2.5.

2.2.2.26 character special file: A file that refers to a device.

One specific type of character special file is a terminal device file, whose access is defined in POSIX.1 {8} section 7.1. Other character special files have no structure defined by this standard, and their use is unspecified by this standard. [POSIX.1 {8}]

2.2.2.27 circumflex: The character ‘‘^’’.

2.2.2.28 collating element: The smallest entity used to determine the logical ordering of strings.

See *collation sequence* (2.2.2.30). A collating element shall consist of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements.

2.2.2.29 collation: The logical ordering of strings according to defined precedence rules.

These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

2.2.2.30 collation sequence: The relative order of collating elements as determined by the setting of the LC_COLLATE category in the current locale.

The character order, as defined for the LC_COLLATE category in the current locale 2
(see 2.5.2.2), defines the relative order of all collating elements, such that each 2
element occupies a unique position in the order. In addition, one or more collation 2
weights can be assigned for each collating element; these weights are used to 2
determine the relative order of strings in, e.g., the `sort` utility. 2

Multilevel sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}. On each level, elements may be
given the same weight (at the primary level, called an *equivalence class*; see 1
2.2.2.47) or be omitted from the sequence. Strings that collate equal using the first assigned weight (primary ordering), are then compared using the next assigned weight (secondary ordering), and so on.

2.2.2.31 column position: A unit of horizontal measure related to characters in a line.

It is assumed that each character in a character set has an intrinsic column width independent of any output device. Each printable character in the portable character set has a column width of one. The standard utilities, when used as described in this standard, assume that all characters have integral column widths. The column width of a character is not necessarily related to the internal representation of the character (numbers of bits or octets).

The column position of a character in a line is defined as one plus the sum of the column widths of the preceding characters in the line. Column positions are numbered starting from 1.

2.2.2.32 command: A directive to the shell to perform a particular task; see 3.9.

2.2.2.33 current working directory: See *working directory* in 2.2.2.159.

2.2.2.34 command language interpreter: See 2.2.2.133.

2.2.2.35 directory: A file that contains directory entries.

No two directory entries in the same directory shall have the same name. [POSIX.1 {8}]

2.2.2.36 directory entry [link]: An object that associates a filename with a file.

Several directory entries can associate names with the same file. [POSIX.1 {8}]

2.2.2.37 dollar-sign: The character ‘‘$’’.

This standard permits the substitution of the ‘‘currency symbol’’ graphic defined in ISO/IEC 646 {1} for this symbol when the character set being used has substituted that graphic for the graphic $. The graphic symbol $ is always used in this standard, but not in any monetary sense.

2.2.2.38 dot: The filename consisting of a single dot character (**.**).

See *pathname resolution* in 2.2.2.104. [POSIX.1 {8}]

In the context of shell special built-in utilities, see 3.14.4.

2.2.2.39 dot-dot: The filename consisting solely of two dot characters (**..**).

See *pathname resolution* in 2.2.2.104. [POSIX.1 {8}]

2.2.2.40 double-quote: The character ‘‘"’’, also known as *quotation-mark*.

2.2.2.41 effective group ID: An attribute of a process that is used in determining various permissions, including file access permissions, described in 2.2.2.55.

See *group ID*. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 3.1.2 (*exec*) and 4.2.2 [*setgid*()]. [POSIX.1 {8}]

2.2.2.42 effective user ID: An attribute of a process that is used in determining various permissions, including file access permissions.

See *user ID*. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 3.1.2 (*exec*) and 4.2.2 [*setuid*()]. [POSIX.1 {8}]

2.2.2.43 empty directory: A directory that contains, at most, directory entries for dot and dot-dot. [POSIX.1 {8}]

2.2.2.44 empty line: A line consisting of only a `<newline>` character.

See also *blank line* (2.2.2.17).

2.2.2.45 empty string [null string]: A character array whose first element is a null character. [POSIX.1 {8}]

2.2.2.46 Epoch: The time 0 hours, 0 minutes, 0 seconds, January 1, 1970, Coordinated Universal Time.

See *seconds since the Epoch*. [POSIX.1 {8}]

2.2.2.47 equivalence class: A set of collating elements with the same primary 1
collation weight. 1

Elements in an equivalence class are typically elements that naturally group together, such as all accented letters based on the same base letter.

The collation order of elements within an equivalence class is determined by the 1
weights assigned on any subsequent levels after the primary weight. 1

2.2.2.48 executable file: A regular file acceptable as a new process image file by the equivalent of the POSIX.1 {8} *exec* family of functions, and thus usable as one form of a utility.

See *exec* in POSIX.1 {8} 3.1.2. The standard utilities described as compilers can produce executable files, but other unspecified methods of producing executable files may also be provided. The internal format of an executable file is unspecified, but a conforming application shall not assume an executable file is a text file.

2.2.2.49 execute: To perform the actions described in 3.9.1.1.

See also *invoke* (2.2.2.79).

2.2.2.50 extended regular expression: A pattern (sequence of characters or symbols) constructed according to the rules defined in 2.8.4.

2.2.2.51 extended security controls: A concept of the underlying system, as follows. [POSIX.1 {8}]

The access control (see *file access permissions*) and privilege (see *appropriate privileges* in 2.2.2.6) mechanisms have been defined to allow implementation-defined extended security controls. These permit an implementation to provide security mechanisms to implement different security policies than described in POSIX.1 {8}. These mechanisms shall not alter or override the defined semantics of any of the functions in POSIX.1 {8}.

2.2.2.52 feature test macro: A `#define`d symbol used to determine whether a particular set of features will be included from a header.

See POSIX.1 {8}2.7.1. [POSIX.1 {8}]

2.2.2.53 FIFO special file [FIFO]: A type of file with the property that data written to such a file is read on a first-in-first-out basis.

Other characteristics of *FIFO*s are described in POSIX.1 {8} 5.3.1 [*open*()], 6.4.1 [*read*()], 6.4.2 [*write*()], and 6.5.3 [*lseek*()]. [POSIX.1 {8}]

2.2.2.54 file: An object that can be written to, or read from, or both.

A file has certain attributes, including access permissions and type. File types include regular file, character special file, block special file, FIFO special file, and directory. Other types of files may be defined by the implementation. [POSIX.1 {8}]

2.2.2.55 file access permissions: A concept of the underlying system, as follows. [POSIX.1 {8}]

The standard file access control mechanism uses the file permission bits, as described below. These bits are set at file creation by *open*(), *creat*(), *mkdir*(), and *mkfifo*() and are changed by *chmod*(). These bits are read by *stat*() or *fstat*().

Implementations may provide *additional* or *alternate* file access control mechanisms, or both. An additional access control mechanism shall only further restrict the access permissions defined by the file permission bits. An alternate access control mechanism shall:

(1) Specify file permission bits for the file owner class, file group class, and file other class of the file, corresponding to the access permissions, to be returned by *stat*() or *fstat*().

(2) Be enabled only by explicit user action, on a per-file basis by the file owner or a user with the appropriate privilege.

(3) Be disabled for a file after the file permission bits are changed for that file with *chmod* (). The disabling of the alternate mechanism need not disable any additional mechanisms defined by an implementation.

Whenever a process requests file access permission for read, write, or execute/search, if no additional mechanism denies access, access is determined as follows:

(1) If a process has the appropriate privilege:

  (a) If read, write, or directory search permission is requested, access is granted.

  (b) If execute permission is requested, access is granted if execute permission is granted to at least one user by the file permission bits or by an alternate access control mechanism; otherwise, access is denied.

(2) Otherwise:

  (a) The file permission bits of a file contain read, write, and execute/search permissions for the file owner class, file group class, and file other class.

  (b) Access is granted if an alternate access control mechanism is not enabled and the requested access permission bit is set for the class (file owner class, file group class, or file other class) to which the process belongs, or if an alternate access control mechanism is enabled and it allows the requested access; otherwise, access is denied.

2.2.2.56 file descriptor: A per-process unique, nonnegative integer used to identify an open file for the purpose of file access. [POSIX.1 {8}]

2.2.2.57 file group class: The property of a file indicating access permissions for a process related to the process's group identification.

A process is in the file group class of a file if the process is not in the file owner class and if the effective group ID or one of the supplementary group IDs of the process matches the group ID associated with the file. Other members of the class may be implementation defined. [POSIX.1 {8}]

2.2.2.58 file hierarchy: A concept of the underlying system, as follows. [POSIX.1 {8}]

Files in the system are organized in a hierarchical structure in which all of the nonterminal nodes are directories and all of the terminal nodes are any other type of file. Because multiple directory entries may refer to the same file, the hierarchy is properly described as a ''directed graph.''

2.2.2.59 file mode: An object containing the file permission bits and other characteristics of a file, as described in POSIX.1 {8} 5.6.1. [POSIX.1 {8}]

2.2.2.60 file mode bits: A file's file permission bits, set-user-ID-on-execution bit (S_ISUID), and set-group-ID-on-execution bit (S_ISGID) (see POSIX.1 {8} 5.6.1.2).

2.2.2.61 filename: A name consisting of 1 to {NAME_MAX} bytes used to name a file.

The characters composing the name may be selected from the set of all character values excluding the slash character and the null character. The filenames dot and dot-dot have special meaning; see *pathname resolution* in 2.2.2.104. A filename is sometimes referred to as a pathname component. [POSIX.1 {8}]

2.2.2.62 filename portability: A concept of the underlying system, as follows. [POSIX.1 {8}]

Filenames should be constructed from the portable filename character set because the use of other characters can be confusing or ambiguous in certain contexts.

2.2.2.63 file offset: The byte position in the file where the next I/O operation begins.

Each open file description associated with a regular file, block special file, or directory has a file offset. A character special file that does not refer to a terminal device may have a file offset. There is no file offset specified for a pipe or FIFO. [POSIX.1 {8}]

2.2.2.64 file other class: The property of a file indicating access permissions for a process related to the process's user and group identification.

A process is in the file other class of a file if the process is not in the file owner class or file group class. [POSIX.1 {8}]

2.2.2.65 file owner class: The property of a file indicating access permissions for a process related to the process's user identification.

A process is in the file owner class of a file if the effective user ID of the process matches the user ID of the file. [POSIX.1 {8}]

2.2.2.66 file permission bits: Information about a file that is used, along with other information, to determine if a process has read, write, or execute/search permission to a file.

The bits are divided into three parts: owner, group, and other. Each part is used with the corresponding file class of processes. These bits are contained in the file mode, as described in POSIX.1 {8} 5.6.1. The detailed usage of the file permission bits in access decisions is described in *file access permissions* in 2.2.2.55. [POSIX.1 {8}]

2.2.2.67 file serial number: A per-file-system unique identifier for a file.

File serial numbers are unique throughout a file system. [POSIX.1 {8}]

2.2.2.68 file system: A collection of files and certain of their attributes.

It provides a name space for file serial numbers referring to those files. [POSIX.1 {8}]

2.2.2.69 file times update: A concept of the underlying system, as follows. [POSIX.1 {8}]

Each file has three distinct associated time values: *st_atime*, *st_mtime*, and *st_ctime*. The *st_atime* field is associated with the times that the file data is accessed; *st_mtime* is associated with the times that the file data is modified; and *st_ctime* is associated with the times that file status is changed. These values are returned in the file characteristics structure, as described in POSIX.1 {8}5.6.1.

Any function in this standard that is required to read or write file data or change the file status indicates which of the appropriate time-related fields are to be ‘‘marked for update.’’ If an implementation of such a function marks for update a time-related field not specified by this standard, this shall be documented, except that any changes caused by pathname resolution need not be documented. For the other functions in this standard (those that are not explicitly required to read or write file data or change file status, but that in some implementations happen to do so), the effect is unspecified.

An implementation may update fields that are marked for update immediately, or it may update such fields periodically. When the fields are updated, they are set to the current time and the update marks are cleared. All fields that are marked for update shall be updated when the file is no longer open by any process, or when a *stat*() or *fstat*() is performed on the file. Other times at which updates are done are unspecified. Updates are not done for files on read-only file systems.

2.2.2.70 file type: See *file* in 2.2.2.54.

2.2.2.71 filter: A command whose operation consists of reading data from standard input or a list of input files and writing data to standard output.

Typically, its function is to perform some transformation on the data stream.

2.2.2.72 foreground process: A process that is a member of a foreground process group. [POSIX.1 {8}]

2.2.2.73 foreground process group: A process group whose member processes have certain privileges, denied to processes in background process groups, when accessing their controlling terminal.

Each session that has established a connection with a controlling terminal has exactly one process group of the session as the foreground process group of that controlling terminal. See POSIX.1 {8} 7.1.1.4. [POSIX.1 {8}]

2.2.2.74 `<form-feed>`: A character that in the output stream shall indicate that printing should start on the next page of an output device.

The `<form-feed>` shall be the character designated by `'\f'` in the C language binding. If `<form-feed>` is not the first character of an output line, the result is unspecified. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next page.

2.2.2.75 group ID: A nonnegative integer, which can be contained in an object of type *gid_t*, that is used to identify a group of system users.

Each system user is a member of at least one group. When the identity of a group is associated with a process, a group ID value is referred to as a real group ID, an effective group ID, one of the (optional) supplementary group IDs, or an (optional) saved set-group-ID. [POSIX.1 {8}]

2.2.2.76 hard link: The relationship between two directory entries that represent the same file; the result of an execution of the `ln` utility or the POSIX.1 {8} *link*() function.

2.2.2.77 home directory: The current directory associated with a user at the time of login.

2.2.2.78 incomplete line: A sequence of text consisting of one or more non-`<newline>` characters at the end of the file.

2.2.2.79 invoke: To perform the actions described in 3.9.1.1, except that searching for shell functions and special built-ins is suppressed.

See also *execute* (2.2.2.49).

2.2.2.80 job control: A facility that allows users to selectively stop (suspend) the execution of processes and continue (resume) their execution at a later point.

The user typically employs this facility via the interactive interface jointly supplied by the terminal I/O driver and a command interpreter. POSIX.1 {8} conforming implementations may optionally support job control facilities; the presence of this option is indicated to the application at compile time or run time by the definition of the {_POSIX_JOB_CONTROL} symbol; see POSIX.1 {8} 2.9. [POSIX.1 {8}]

2.2.2.81 line: A sequence of text consisting of zero or more non-`<newline>` characters plus a terminating `<newline>` character.

2.2.2.82 link: See *directory entry* in 2.2.2.36.

2.2.2.83 link count: The number of directory entries that refer to a particular file. [POSIX.1 {8}]

2.2.2.84 locale: The definition of the subset of a user's environment that depends on language and cultural conventions; see 2.5.

2.2.2.85 login: The unspecified activity by which a user gains access to the system.

Each login shall be associated with exactly one login name. [POSIX.1 {8}]

2.2.2.86 login name: A user name that is associated with a login. [POSIX.1 {8}]

2.2.2.87 mode: A collection of attributes that specifies a file's type and its access permissions.

See *file access permissions* in 2.2.2.55. [POSIX.1 {8}]

2.2.2.88 multicharacter collating element: A sequence of two or more characters that collate as an entity.

For example, in some coded character sets, an accented character is represented by a (nonspacing) accent, followed by the letter. Another example is the Spanish elements ''ch''and ''ll.''

2.2.2.89 negative response: An input string that matches one of the responses acceptable to the LC_MESSAGES category keyword `noexpr`, matching an extended regular expression in the current locale.

See 2.5.

2.2.2.90 `<newline>`: A character that in the output stream shall indicate that 1
printing should start at the beginning of the next line.

The `<newline>` shall be the character designated by `'\n'` in the C language binding. It is unspecified whether this character is the exact sequence transmitted to an output device by the system to accomplish the movement to the next line.

2.2.2.91 NUL: A character with all bits set to zero.

2.2.2.92 null string: See *empty string* in 2.2.2.45.

2.2.2.93 number-sign: The character '#''.

This standard permits the substitution of the ''pound sign'' graphic defined in ISO/IEC 646 {1} for this symbol when the character set being used has substituted that graphic for the graphic #. The graphic symbol # is always used in this standard.

2.2.2.94 object file: A regular file containing the output of a compiler, formatted as input to a linkage editor for linking with other object files into an executable form.

The methods of linking are unspecified and may involve the dynamic linking of objects at run-time. The internal format of an object file is unspecified, but a conforming application shall not assume an object file is a text file.

2.2.2.95 open file: A file that is currently associated with a file descriptor. [POSIX.1 {8}]

2.2.2.96 operand: An argument to a command that is generally used as an object supplying information to a utility necessary to complete its processing.

Operands generally follow the options in a command line. See 2.10.1.

2.2.2.97 option: An argument to a command that is generally used to specify changes in the *utility*'s default behavior; see 2.10.1.

2.2.2.98 option-argument: A parameter that follows certain options.

In some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. See 2.10.1.

2.2.2.99 parent directory:

(1) When discussing a given directory, the directory that both contains a directory entry for the given directory and is represented by the pathname dot-dot in the given directory.

(2) When discussing other types of files, a directory containing a directory entry for the file under discussion.

This concept does not apply to dot and dot-dot. [POSIX.1 {8}]

2.2.2.100 parent process: See *process* in 2.2.2.114. [POSIX.1 {8}]

2.2.2.101 parent process ID: An attribute of a new process after it is created by a currently active process.

The parent process ID of a process is the process ID of its creator, for the lifetime of the creator. After the creator's lifetime has ended, the parent process ID is the process ID of an implementation-defined system process. [POSIX.1 {8}]

2.2.2.102 pathname: A string that is used to identify a file.

A pathname consists of, at most, {PATH_MAX} bytes, including the terminating null character. It has an optional beginning slash, followed by zero or more filenames separated by slashes. If the pathname refers to a directory, it may also have one or more trailing slashes. Multiple successive slashes are considered to be the same as one slash. A pathname that begins with two successive slashes may be interpreted in an implementation-defined manner, although more than two leading slashes shall be treated as a single slash. The interpretation of the pathname is described in *pathname resolution* in 2.2.2.104. [POSIX.1 {8}]

2.2.2.103 pathname component: See *filename* in 2.2.2.61. [POSIX.1 {8}]

2.2.2.104 pathname resolution: A concept of the underlying system, as follows. [POSIX.1 {8}]

Pathname resolution is performed for a process to resolve a pathname to a particular file in a file hierarchy. There may be multiple pathnames that resolve to the same file.

Each filename in the pathname is located in the directory specified by its predecessor (for example, in the pathname fragment `a/b`, file `b` is located in directory `a`). Pathname resolution fails if this cannot be accomplished. If the pathname begins with a slash, the predecessor of the first filename in the pathname is taken to be the root directory of the process (such pathnames are referred to as absolute pathnames). If the pathname does not begin with a slash, the predecessor of the first filename of the pathname is taken to be the current working directory of the process (such pathnames are referred to as "relative pathnames").

The interpretation of a pathname component is dependent on the values of {NAME_MAX} and {_POSIX_NO_TRUNC} associated with the path prefix of that component. If any pathname component is longer than {NAME_MAX}, and {_POSIX_NO_TRUNC} is in effect for the path prefix of that component [see *pathconf*() in POSIX.1 {8} 5.7.1], the implementation shall consider this an error condition. Otherwise, the implementation shall use the first {NAME_MAX} bytes of the pathname component.

The special filename dot refers to the directory specified by its predecessor. The special filename dot-dot refers to the parent directory of its predecessor directory. As a special case, in the root directory, dot-dot may refer to the root directory itself.

A pathname consisting of a single slash resolves to the root directory of the process. A null pathname is invalid.

2.2.2.105 path prefix: A pathname, with an optional ending slash, that refers to a directory. [POSIX.1 {8}]

2.2.2.106 pattern: A sequence of characters used either with regular expression notation (see 2.8) or for pathname expansion (see 3.6.6), as a means of selecting various character strings or pathnames, respectively.

The syntaxes of the two patterns are similar, but not identical; this standard always indicates the type of pattern being referred to in the immediate context of the use of the term.

2.2.2.107 period: The character ''**.**''.

The term *period* is contrasted against *dot* (2.2.2.38), which is used to describe a specific directory entry.

2.2.2.108 permissions: See *file access permissions* in 2.2.2.55.

2.2.2.109 pipe: An object accessed by one of the pair of file descriptors created by the POSIX.1 {8} *pipe*() function.

Once created, the file descriptors can be used to manipulate it, and it behaves identically to a FIFO special file when accessed in this way. It has no name in the file hierarchy. [POSIX.1 {8}]

2.2.2.110 portable character set: The set of characters described in 2.4 that is supported on all conforming systems.

This term is contrasted against the smaller *portable filename character set*; see 2.2.2.111.

2.2.2.111 portable filename character set: The set of characters from which portable filenames are constructed.

For a filename to be portable across conforming implementations of this standard, it shall consist only of the following characters:

```
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
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
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore, and hyphen characters, respectively. The hyphen shall not be used as the first character of a portable filename. Upper- and lowercase letters shall retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used. [POSIX.1 {8}]

2.2.2.112 printable character: One of the characters included in the `print` character classification of the LC_CTYPE category in the current locale; see 2.5.2.1.

2.2.2.113 privilege: See *appropriate privileges* in 2.2.2.6. [POSIX.1 {8}]

2.2.2.114 process: An address space and single thread of control that executes within that address space, and its required system resources.

A process is created by another process issuing the POSIX.1 {8} *fork*() function. The process that issues *fork*() is known as the parent process, and the new process created by the *fork*() is known as the child process. [POSIX.1 {8}]

The attributes of processes required by POSIX.2 form a subset of those in POSIX.1 {8}; see 2.9.1.

2.2.2.115 process group: A collection of processes that permits the signaling of related processes.

Each process in the system is a member of a process group that is identified by a process group ID. A newly created process joins the process group of its creator. [POSIX.1 {8}]

2.2.2.116 process group ID: The unique identifier representing a process group during its lifetime.

A process group ID is a positive integer that can be contained in a *pid_t*. It shall not be reused by the system until the process group lifetime ends. [POSIX.1 {8}]

2.2.2.117 process group leader: A process whose process ID is the same as its process group ID. [POSIX.1 {8}]

2.2.2.118 process ID: The unique identifier representing a process.

A process ID is a positive integer that can be contained in a *pid_t*. A process ID shall not be reused by the system until the process lifetime ends. In addition, if there exists a process group whose process group ID is equal to that process ID, the process ID shall not be reused by the system until the process group lifetime ends. A process that is not a system process shall not have a process ID of 1. [POSIX.1 {8}]

2.2.2.119 program: A prepared sequence of instructions to the system to accomplish a defined task.

The term *program* in POSIX.2 encompasses applications written in the Shell Command Language, complex utility input languages (for example, `awk`, `lex`, `sed`, etc.), and high-level languages.

2.2.2.120 read-only file system: A file system that has implementation-defined characteristics restricting modifications. [POSIX.1 {8}]

2.2.2.121 real group ID: The attribute of a process that, at the time of process creation, identifies the group of the user who created the process.

See *group ID* in 2.2.2.75. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 4.2.2 [*setgid*()]. [POSIX.1 {8}]

2.2.2.122 real user ID: The attribute of a process that, at the time of process creation, identifies the user who created the process.

See *user ID* in 2.2.2.154. This value is subject to change during the process lifetime, as described in POSIX.1 {8} 4.2.2 [*setuid*()]. [POSIX.1 {8}]

2.2.2.123 regular expression: A pattern (sequence of characters or symbols) 1
constructed according to the rules defined in 2.8. 1

2.2.2.124 regular file: A file that is a randomly accessible sequence of bytes, with no further structure imposed by the system. [POSIX.1 {8}]

2.2.2.125 relative pathname: See *pathname resolution* in 2.2.2.104. [POSIX.1 {8}]

2.2.2.126 root directory: A directory, associated with a process, that is used in pathname resolution for pathnames that begin with a slash. [POSIX.1 {8}]

2.2.2.127 saved set-group-ID: An attribute of a process that allows some flexibility in the assignment of the effective group ID attribute, when the saved set-user-ID option is implemented, as described in POSIX.1 {8} 3.1.2 (*exec*) and 4.2.2 [*setgid*()]. [POSIX.1 {8}]

2.2.2.128 saved set-user-ID: An attribute of a process that allows some flexibility in the assignment of the effective user ID attribute, when the saved set-user-ID option is implemented, as described in POSIX.1 {8} 3.1.2 and 4.2.2 [*setuid*()]. [POSIX.1 {8}]

2.2.2.129 seconds since the Epoch: A value to be interpreted as the number of seconds between a specified time and the Epoch.

A Coordinated Universal Time name (specified in terms of seconds (*tm_sec*), minutes (*tm_min*), hours (*tm_hour*), days since January 1 of the year (*tm_yday*), and calendar year minus 1900 (*tm_year*) is related to a time represented as seconds since the Epoch, according to the expression below.

If the year < 1970 or the value is negative, the relationship is undefined. If the year ≥ 1970 and the value is nonnegative, the value is related to a Coordinated Universal Time name according to the expression:

$$tm\_sec + tm\_min * 60 + tm\_hour * 3\,600 + tm\_yday * 86\,400 +$$
$$(tm\_year - 70) * 31\,536\,000 + ((tm\_year - 69)/4) * 86\,400$$

[POSIX.1 {8}]

2.2.2.130 session: A collection of process groups established for job control purposes.

Each process group is a member of a session. A process is considered to be a member of the session of which its process group is a member. A newly created process joins the session of its creator. A process can alter its session membership (see POSIX.1 {8} 4.3.2 [*setsid*()]. Implementations that support the POSIX.1 {8} *setpgid*() function (see POSIX.1 {8} 4.3.3) can have multiple process groups in the same session. [POSIX.1 {8}]

2.2.2.131 session leader: A process that has created a session; see POSIX.1 {8} 4.3.2 [*setsid*()]. [POSIX.1 {8}]

2.2.2.132 session lifetime: The period between when a session is created and the end of the lifetime of all the process groups that remain as members of the session. [POSIX.1 {8}]

2.2.2.133 shell: A program that interprets sequences of text input as commands.

It may operate on an input stream or it may interactively prompt and read commands from a terminal.

2.2.2.134 Shell, The: The Shell Command Language Interpreter (see 4.56), a specific instance of a shell.

2.2.2.135 shell script: A file containing shell commands.

If the file is made executable, it can be executed by specifying its name as a simple command (see the description of *simple command* in 3.9.1). Execution of a shell script causes a shell to execute the commands within the script. Alternately, a shell can be requested to execute the commands in a shell script by specifying the name of the shell script as the operand to the `sh` utility.

2.2.2.136 signal: A mechanism by which a process may be notified of, or affected by, an event occurring in the system.

Examples of such events include hardware exceptions and specific actions by processes. The term *signal* is also used to refer to the event itself. [POSIX.1 {8}]

2.2.2.137 single-quote: The character ‘‘'’’, also known as *apostrophe*.

2.2.2.138 slash: The character ''/'', also known as *solidus* .

2.2.2.139 source code: When dealing with the Shell Command Language, source code is input to the command language interpreter.

The term *shell script* is synonymous with this meaning.

When dealing with the C Language Bindings Option, source code is input to a C compiler conforming to the C Standard {7}.

When dealing with another ISO/IEC conforming language, source code is input to a compiler conforming to that ISO/IEC standard.

Source code also refers to the input statements prepared for the following standard utilities: `awk`, `bc`, `ed`, `lex`, `localedef`, `make`, `sed`, and `yacc`.

Source code can also refer to a collection of sources meeting any or all of these meanings.

2.2.2.140 `<space>`: The character defined in 2.4 as `<space>`.

The `<space>` character is a member of the `space` character class of the current locale, but represents the single character, and not all of the possible members of the class. (See 2.2.2.158.)

2.2.2.141 standard error: An output stream usually intended to be used for diagnostic messages.

2.2.2.142 standard input: An input stream usually intended to be used for primary data input.

2.2.2.143 standard output: An output stream usually intended to be used for primary data output.

2.2.2.144 standard utilities: The utilities defined by this standard, in the Sections 4, 5, and 6, and Annex A, and Annex C, and in similar sections of utility definitions introduced in future revisions of, and supplements to, this standard.

2.2.2.145 stream: An ordered sequence of characters, as described by the C Standard {7}.

2.2.2.146 supplementary group ID: An attribute of a process used in determining file access permissions.

A process has up to {NGROUPS_MAX} supplementary group IDs in addition to the effective group ID. The supplementary group IDs of a process are set to the supplementary group IDs of the parent process when the process is created. Whether a process's effective group ID is included in or omitted from its list of supplementary group IDs is unspecified. [POSIX.1 {8}]

2.2.2.147 system: An implementation of this standard.

2.2.2.148 <tab>: The horizontal tab character.

2.2.2.149 terminal [terminal device]: A character special file that obeys the specifications of the POSIX.1 {8} General Terminal Interface. [POSIX.1 {8}]

2.2.2.150 text column: A roughly rectangular block of characters capable of being laid out side-by-side next to other text columns on an output page or terminal screen.

The widths of text columns are measured in column positions.

2.2.2.151 text file: A file that contains characters organized into one or more lines.

The lines shall not contain NUL characters and none shall exceed {LINE_MAX} bytes in length, including the <newline>. Although POSIX.1 {8} does not distinguish between text files and binary files (see the C Standard {7}), many utilities only produce predictable or meaningful output when operating on text files. The standard utilities that have such restrictions always specify *text files* in their Standard Input or Input Files subclauses.

2.2.2.152 tilde: The character '~'.

2.2.2.153 user database: See Section 9 in POSIX.1 {8}.

2.2.2.154 user ID: A nonnegative integer, which can be contained in an object of type *uid_t*, that is used to identify a system user.

When the identity of a user is associated with a process, a user ID value is referred to as a real user ID, an effective user ID, or an (optional) saved set-user-ID. [POSIX.1 {8}]

2.2.2.155 user name: A string that is used to identify a user, as described in POSIX.1 {8} 9.1. [POSIX.1 {8}]

2.2.2.156 utility: A program that can be called by name from a shell to perform a specific task, or related set of tasks.

This program shall either be an executable file, such as might be produced by a compiler/linker system from computer source code, or a file of shell source code, directly interpreted by the shell. The program may have been produced by the user, provided by the implementor of this standard, or acquired from an independent distributor. The term *utility* does not apply to the special built-in utilities provided as part of the shell command language; see 3.14. The system may implement certain utilities as shell functions (see 3.9.5) or built-ins (see 2.3), but only an application that is aware of the command search order described in 3.9.1.1 or of performance characteristics can discern differences between the behavior of such a function or built-in and that of a true executable file.

2.2.2.157 `<vertical-tab>`: The vertical tab character.

2.2.2.158 white space: A sequence of one or more characters that belong to the `space` character class as defined via the LC_CTYPE category in the current locale.

In the POSIX Locale, white space consists of one or more `<blank>`s (`<space>`s and `<tab>`s), `<newline>`s, `<carriage-return>`s, `<form-feed>`s, and `<vertical-tab>`s.

2.2.2.159 working directory [current working directory]: A directory, associated with a process, that is used in pathname resolution for pathnames that do not begin with a slash.

2.2.2.160 write: To output characters to a file, such as standard output or standard error.

Unless otherwise stated, standard output is the default output destination for all uses of the term *write*.

2.2.2.161 General Terms Rationale. *(This subclause is not a part of P1003.2)*

Many of the terms originated in POSIX.1 {8} and are duplicated in this standard to meet editorial requirements. In some cases, there is supplementary text that presents additional information concerning POSIX.2 aspects of the concept.

This standard uses the term *character* to mean a sequence of one or more bytes
representing a single graphic symbol, as defined in POSIX.1 {8}. The deviation in 1
the exact text of the C Standard {7} definition for *byte* meets the intent of the 1
C Standard {7} Rationale and the developers of POSIX.1 {8}, but clears up the 1
ambiguity raised by the term *basic execution character set*, which is not defined in 1
POSIX.1 {8}. It is expected that a future version of POSIX.1 {8} will align with the 1
text used here. The octet-minimum requirement is merely a reflection of the 1
{CHAR_BIT} value in POSIX.1 {8} and the C Standard {7}. 1

The POSIX.1 {8} term *file mode* is a superset of the POSIX.2 *file mode bits*. POSIX.1 {8} defines the file mode as the entire *mode_t* object (which includes the file type in historically the upper four bits, the sticky bit on most implementations, and potentially other nonstandardized attributes), while POSIX.2 file mode bits include only the eleven defined bits.

The terms *command* and *utility* are related but have distinct meanings. Command is defined as ‘‘a directive to a shell to perform a specific task.’’ The directive can be in the form of a single utility name (for example, `ls`), or the directive can take the form of a compound command (for example, `ls | grep name | pr`).

A utility is a program that is callable by name from a shell. Issuing only the utility's name to a shell is the equivalent of a one-word command. A utility may be invoked as a separate program that executes in a different process than the command language interpreter, or may be implemented as a part of the command language interpreter. For example, the `echo` command (the directive to perform a specific task) may be implemented such that the `echo` utility (the logic that performs the task of echoing) is in a separate program; and therefore, is executed in a process that is different than the command language interpreter. Conversely, the logic that performs the `echo` utility could be built into the command language interpreter; and therefore, execute in the same process as the command language interpreter.

The terms *tool* and *application* can be thought of as being synonymous with *utility* from the perspective of the operating system kernel. Tools, applications, and utilities have historically run, typically, in processes above the kernel level. Tools and utilities have been historically a part of the operating system nonkernel code, and performed system related functions such as listing directory contents, checking file systems, repairing file systems, or extracting system status information. Applications have not generally been a part of the operating system, and perform nonsystem related functions such as word processing, architectural design, mechanical design, workstation publishing, or financial analysis. Utilities have most frequently been provided by the operating system vendor, applications by third party software vendors or by the users themselves. Nevertheless, the standard does not differentiate between tools, utilities, and applications when it comes to receiving services from the system, a shell, or the standard utilities. (For example, the `xargs` utility invokes another utility; it would be of fairly limited usefulness if the users couldn't run their own applications in place of the standard utilities.) Utilities are not applications in the sense that they are not themselves subjects to the restrictions of this standard or any other standard— there is no requirement for `grep`, `stty`, or any of the utilities defined here to be any of the classes of Conforming POSIX.2 Applications.

The term *text file* does not prevent the inclusion of control or other nonprintable characters (other than NUL). Therefore, standard utilities that list text files as inputs or outputs are either able to process the special characters gracefully or they explicitly describe their limitations within their individual subclauses. The definition of *text file* has caused a good deal of controversy. The only difference between text and binary here is that text files have lines of (less than {LINE_MAX}) bytes, with no NUL characters, each terminated by a `<newline>` character. The definition allows a file with a single `<newline>`, but not a totally empty file, to be called a text file. If a file ends with an incomplete line it is not strictly a text file by this definition. A related point is that the `<newline>` character referred to in this standard is not some generic line separator, but a single character; files created on systems where they use multiple characters for ends of lines are not portable to all POSIX systems without some translation process unspecified by this standard.

The term *hard link* is historically-derived. In systems without extensions to `ln`, it is a synonym for *link*. The concept of a *symbolic link* originated with BSD systems and the term *hard* is used to differentiate between the two types of links.

There are some terms used that are undefined in POSIX.2, POSIX.1 {8}, or the C Standard {7}. The working group believes that these terms have a ''common usage,'' and that a definition in POSIX.2 would not be appropriate. Terms in this category include, but are not limited to, the following: *application, character set, login session, user*. Good sources for general terms of this type are the *ISO / AFNOR Dictionary of Computer Science* {B12} and *IEEE Dictionary* {B18}.

The term *file name* was defined in previous drafts to be a synonym for *pathname*. It was removed in the face of objections that it was too close to *filename*, which means something different (a pathname component). The general solution to this has been to use the term *file* in parameter names, rather than *file_name*, and to make more liberal use of the correct term, *pathname*; an alternate solution has been to replace *file name* with *the name of the file*.

Many character names are included in this subclause. Because of historical usage, some of these names are a bit different than the ones used in international standards for character sets, such as ISO/IEC 646 {1}. It was felt that many more UNIX system people than character set lawyers would be reading and reviewing the standard, so the former group was the one accommodated. On the other hand, the precise definitions of <space>, <blank>, and *white space* have replaced common usage (where they have been used virtually interchangeably), as the standard attempts to balance readability against precision.

In earlier drafts, the names for the character pairs ( ), [ ], and { } were referred to as ''opening'' and ''closing'' parentheses, brackets, and braces. These were changed to the current ''left'' and right.'' When the characters are used to express natural language, the terms ''open'' and ''close'' imply text direction more strongly than ''left'' and ''right.'' By POSIX.2 definition, the character <open-parenthesis> will always be mapped to the glyph '(' regardless of the locale. But when reading right-to-left, the opening punctuation of a parenthesized text segment would be ')'. The <left-parenthesis> and <right-parenthesis> forms are the correct ones because the punctuation appears on the left and right, respectively, of the parenthesized text regardless of the direction one might be reading the text.

The <backspace> character and the ERASE special character defined in POSIX.1 {8} should not be confused. The use of the <backspace> character and the ERASE special character defined in the POSIX.1 {8} *termios* clause on special characters (7.1.1.9) are distinct even though the ERASE special character may be set to <backspace>.

In most one-byte character sets, such as ASCII, the concepts of column positions is identical to character positions and to bytes. Therefore, it has been historically acceptable for some implementations to describe line folding or tab stops or table column alignment in terms of bytes or character positions. Other character sets pose complications, as they can have internal representations longer than one octet and they can have displayable characters that have different widths on the terminal screen or printer.

In this standard the term *column positions* has been defined to mean character— not byte—positions in input files (such as ''column position 7 of the FORTRAN input''). Output files describe the column position in terms of the display width of the narrowest printable character in the character set, adjusted to fit the characteristics of the output device. It is very possible that *n* column positions will not be able to hold *n* characters in some character sets, unless all of those characters are of the narrowest width. It is assumed that the implementation is aware of the width of the various characters, deriving this information from the value of **LC_CTYPE**, and thus can determine how many column positions to allot for each character in those utilities where it is important. This information is not available to the portable application writer because POSIX.2 provides no interface specification to retrieve such information.

The term *column position* was used instead of the more natural *column* as the latter is frequently used in the standard in the different contexts of columns of figures, columns of table values, etc. Wherever confusion might result, these latter types of columns are referred to as *text columns*.

The definition of *binary file* was removed, as the term is not used in the standard.

The ISO/IEC 646 {1} character set standard permits substitution of national currency symbols for the character $ in the ''reference character set'' (which is the same as ASCII). This standard permits the substitution only of the actual characters shown in ISO/IEC 646 {1}: currency sign for the dollar sign and pound sign for the number sign. This document uses the latter names and their symbols, but it is valid for an implementation to accept, for instance, the pound sign (£) as a comment character in the shell, if that is what the locale's character set uses instead of the number sign (#). Other variation of national currency symbols are not allowed, per the request of the WG15 POSIX working group.

The term *stream* is not related to System V's STREAMS communications facility; it is derived from historical UNIX system usage and has been made official by the C Standard {7}. The POSIX.2 standard makes no differentiation between C's *text stream* and *binary stream*.

The formula used in the POSIX.1 {8} definition of *seconds since the Epoch* is not 1
perfect in all cases. See the related rationale in POSIX.1 {8}. 1

2.2.3 Abbreviations

For the purposes of this standard, the following abbreviations apply:

2.2.3.1 C Standard: ISO/IEC 9899: . . . , *Information processing systems—Programming languages—C* {7}.

2.2.3.2 ERE: An Extended Regular Expression, as defined in 2.8.4.

2.2.3.3 LC_*: An abbreviation used to represent all of the environment variables named in 2.6 whose names begin with the characters 'LC_''.

2.2.3.4 POSIX.1: ISO/IEC 9945-1: 1990: *Information technology—Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API) [C Language]* {8}.

2.2.3.5 POSIX.2: This standard.

2.2.3.6 RE [BRE]: A Basic Regular Expression, as defined in 2.8.3.

2.3 Built-in Utilities

Any of the standard utilities may be implemented as *regular built-in* utilities within the command language interpreter. This is usually done to increase the performance of frequently-used utilities or to achieve functionality that would be more difficult in a separate environment. The utilities named in Table 2-2 are frequently provided in built-in form. All of the utilities named in the table have special properties in terms of command search order within the shell, as described in 3.9.1.1.

Table 2-2 – Regular Built-in Utilities

`cd`	`false`	`kill`	`true`	`wait`
`command`	`getopts`	`read`	`umask`	

However, all of the standard utilities, including the regular built-ins in the table, but not the special built-ins described in 3.14, shall be implemented in a manner so that they can be accessed via the POSIX.1 {8} *exec* family of functions (if the underlying operating system provides the services of such a family to application programs) and can be invoked directly by those standard utilities that require it (`env`, `find`, `nohup`, `xargs`).

Since versions shall be provided for all utilities except for those listed previously, an application running on a system that conforms to both POSIX.1 {8} and Section 7 of this standard can use the *exec* family of functions, in addition to the shell command interface in 7.1 [such as the *system* () and *popen* () functions in the C binding] defined by this standard, to execute any of these utilities.

2.3.1 Built-in Utilities Rationale. *(This subclause is not a part of P1003.2)*

In earlier drafts, the table of built-ins implied two things to a conforming application: these may be built-ins and these need not be executable. The second implication has now been removed and all utilities can be *exec*-ed. There is no requirement that these be actually built into the shell itself, but many shells will want to do so because 3.9.1.1 requires that they be found prior to the **PATH** search. The shell could satisfy its requirements by keeping a list of the names and directly accessing the file-system versions regardless of **PATH**. Providing all of the required functionality for those such as `cd` or `read` would be more difficult.

There were originally three justifications for allowing the omission of *exec*-able versions:

(1) This would require wasting space in the file system, at the expense of very small systems. However, it has been pointed out that all nine in the table can be provided with nine links to a single-line shell script:

```
$0 "$@"
```

(2) There is no sense in requiring invocation of utilities like `cd` because they have no value outside the shell environment or cannot be useful in a child process. However, counter-examples always seemed to be available for even the strangest cases:

```
find . -type d -exec cd {} ; -exec foo {} ;
```
(which invokes `foo` on accessible directories)

```
ps . . . | sed . . . | xargs kill
```

```
find . -exec true ; -a . . .
```
(where `true` is used for temporary debugging)

(3) It is confusing to have something such as `kill` that can easily be in the file system in the base standard, but requires built-in status for the UPE (for the `%` job control job ID notation). It was decided that it was more appropriate to describe the required functionality (rather than the implementation) to the system implementors and let them decide how to satisfy it.

On the other hand, there were objections raised during balloting that any distinction like this between utilities was not useful to applications and that the cost to correct it was small. These arguments were ultimately the most effective.

There were varying reasons for including utilities in the table of built-ins:

`cd`, `getopts`, `read`, `umask`, `wait`
: The functionality of these utilities is performed more simply within the context of the current process. An example can be taken from the usage of the `cd` utility. The purpose of the utility is to change the working directory for subsequent operations. The actions of `cd` affect the process in which `cd` is executed and all subsequent child processes of that process. Based on the POSIX.1 {8} process model, changes in the process environment of a child process have no effect on the parent process. If the `cd` utility were executed from a child process, the working directory change would be effective only in the child process. Child processes initiated subsequent to the child process that executed the `cd` utility would not have a changed working directory relative to the parent process.

`command`
: This utility was placed in the table primarily to protect scripts that are concerned about their **PATH** being manipulated. The ‘‘secure’’ shell script example in 4.12.10 would not be possible if a **PATH** change retrieved an alien version of `command`. (An alternative would have been to implement `getconf` as a built-in, but it was felt that it carried too many changing configuration strings to require in the shell.)

`kill`
: Since common extensions to `kill` (including the planned User Portability Extension) provide optional job control functionality using shell notation (`%1`, `%2`, etc.), some implementations would

find it extremely difficult to provide this outside the shell.

`true`, `false`

These are in the table as a courtesy to programmers who wish to use the ‘`while true`’’ shell construct without protecting `true` from **PATH** searches. (It is acknowledged that ‘`while :`’’ also works, but the idiom with `true` is historically pervasive.)

All utilities, including those in the table, are accessible via the functions in 7.1.1 or 7.1.2 [such as *system* () or *popen* ()]. There are situations where the return functionality of *system* () and *popen* () is not desirable. Applications that require the exit status of the invoked utility will not be able to use *system* () or *popen* (), since the exit status returned is that of the command language interpreter rather than that of the invoked utility. The alternative for such applications is the use of the *exec* family. (The text concerning conformance to POSIX.1 {8} was included because where *exec* is not provided in the underlying system, there is no way to require that utilities be *exec*-able).

2.4 Character Set

Table 2-3 – Character Set and Symbolic Names

Symbolic Name	Glyph	Symbolic Name	Glyph	Symbolic Name	Glyph
<NUL>		<colon>	:	<circumflex>	^
<alert>		<semicolon>	;	<circumflex-accent>	^
<backspace>		<less-than-sign>	<	<underscore>	_
<tab>		<equals-sign>	=	<low-line>	_
<newline>		<greater-than-sign>	>	<grave-accent>	`
<vertical-tab>		<question-mark>	?	<a>	a
<form-feed>		<commercial-at>	@	<b>	b
<carriage-return>		<A>	A	<c>	c
<space>		<B>	B	<d>	d
<exclamation-mark>	!	<C>	C	<e>	e
<quotation-mark>	"	<D>	D	<f>	f
<number-sign>	#	<E>	E	<g>	g
<dollar-sign>	$	<F>	F	<h>	h
<percent-sign>	%	<G>	G	<i>	i
<ampersand>	&	<H>	H	<j>	j
<apostrophe>	'	<I>	I	<k>	k
<left-parenthesis>	(	<J>	J	<l>	l
<right-parenthesis>	)	<K>	K	<m>	m
<asterisk>	*	<L>	L	<n>	n
<plus-sign>	+	<M>	M	<o>	o
<comma>	,	<N>	N	<p>	p
<hyphen>	-	<O>	O	<q>	q
<hyphen-minus>	-	<P>	P	<r>	r
<period>	.	<Q>	Q	<s>	s
<full-stop>	.	<R>	R	<t>	t
<slash>	/	<S>	S	<u>	u
<solidus>	/	<T>	T	<v>	v
<zero>	0	<U>	U	<w>	w
<one>	1	<V>	V	<x>	x
<two>	2	<W>	W	<y>	y
<three>	3	<X>	X	<z>	z
<four>	4	<Y>	Y	<left-brace>	{
<five>	5	<Z>	Z	<left-curly-bracket>	{
<six>	6	<left-square-bracket>	[	<vertical-line>	|
<seven>	7	<backslash>	\	<right-brace>	}
<eight>	8	<reverse-solidus>	\	<right-curly-bracket>	}
<nine>	9	<right-square-bracket>	]	<tilde>	~

Conforming implementations shall support one or more coded character sets. Each supported coded character set shall include the *portable character set* specified in Table 2-3. The table defines the characters in the portable character set and the corresponding symbolic character names used to identify each character in a character set description file. The names are chosen to correspond closely with character names defined in other international standards. The table contains more than one symbolic character name for characters whose traditional name differs from the chosen name.

This standard places only the following requirements on the encoded values of the characters in the portable character set:

(1) If the encoded values associated with each member of the portable character set are not invariant across all locales supported by the implementation, the results achieved by an application accessing those locales are unspecified.

(2) The encoded values associated with the digits `'0'` to `'9'` shall be such that the value of each character after `'0'` shall be one greater than the value of the previous character.

(3) A null character, NUL, which has all bits set to zero, shall be in the set of characters.

Conforming implementations shall support certain character and character set attributes, as defined in 2.5.1.

2.4.1 Character Set Description File

Implementations shall provide a character set description file for at least one coded character set supported by the implementation. These files are referred to elsewhere in this standard as *charmap* files. It is implementation defined whether or not users or applications can provide additional character set description files. If such a capability is supported, the system documentation shall describe the rules for the creation of such files.

Each character set description file shall define characteristics for the coded character set and the encoding for the characters specified in Table 2-3, and may define encoding for additional characters supported by the implementation. Other information about the coded character set may also be in the file. Coded character set character values shall be defined using symbolic character names followed by character encoding values.

Each symbolic name specified in Table 2-3 shall be included in the file and shall
be mapped to a unique encoding value (except for those symbolic names that are 1
shown with identical glyphs). If the control characters commonly associated with 1
the symbolic names in Table 2-4 are supported by the implementation, the symbolic names and their corresponding encoding values shall be included in the file. 1
Some of the values associated with the symbolic names in this table also may be 1
contained in Table 2-3. 1

The following declarations can precede the character definitions. Each shall consist of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one of more `<blank>`s, followed by the value to be assigned to the symbol.

`<code_set_name>` The name of the coded character set for which the character set description file is defined. The characters of the
name shall be taken from the set of characters with 1

Table 2-4 – Control Character Set

<ACK>	<DC2>	<ENQ>	<FS>	<IS4>	<SOH>	1
<BEL>	<DC3>	<EOT>	<GS>	<LF>	<STX>	1
<BS>	<DC4>	<ESC>	<HT>	<NAK>	<SUB>	1
<CAN>	<DEL>	<ETB>	<IS1>	<RS>	<SYN>	1
<CR>	<DLE>	<ETX>	<IS2>	<SI>	<US>	1
<DC1>	<EM>	<FF>	<IS3>	<SO>	<VT>	1

visible glyphs defined in Table 2-3. 1

`<mb_cur_max>` The maximum number of bytes in a multibyte character. This shall default to 1.

`<mb_cur_min>` An unsigned positive integer value that shall define the minimum number of bytes in a character for the encoded character set. The value shall be less than or equal to `mb_cur_max`. If not specified, the minimum number shall be equal to `mb_cur_max`.

`<escape_char>` The escape character used to indicate that the characters following shall be interpreted in a special way, as defined later in this subclause. This shall default to backslash (\), which is the character glyph used in all the following text and examples, unless otherwise noted.

`<comment_char>` The character, that when placed in column 1 of a charmap line, is used to indicate that the line shall be ignored. The default character shall be the number-sign (#).

The character set mapping definitions shall be all the lines immediately following an identifier line containing the string `CHARMAP` starting in column 1, and preceding a trailer line containing the string `END CHARMAP` starting in column 1. Empty lines and lines containing a `comment_char` in the first column shall be ignored. Each noncomment line of the character set mapping definition (i.e., between the `CHARMAP` and `END CHARMAP` lines of the file) shall be in either of two forms:

`"%s %s %s\n"`, *<symbolic-name>* , *<encoding>* , *<comments>*

or

`"%s. . .%s %s %s\n"`, *<symbolic-name>* , *<symbolic-name>* , *<encoding>* , *<comments>*

In the first format, the line in the character set mapping definition defines a single symbolic name and a corresponding encoding. A symbolic name is one or more characters from the set shown with visible glyphs in Table 2-3, enclosed between angle brackets. A character following an escape character shall be interpreted as itself; for example, the sequence ‘`<\\\>>`’’ represents the symbolic name ‘`\>`’’

enclosed between angle brackets.

In the second format, the line in the character set mapping definition defines a range of one or more symbolic names. In this form, the symbolic names shall consist of zero or more nonnumeric characters from the set shown with visible glyphs in Table 2-3, followed by an integer formed by one or more decimal digits. The characters preceding the integer shall be identical in the two symbolic names, and the integer formed by the digits in the second symbolic name shall be equal to or greater than the integer formed by the digits in the first name. This shall be interpreted as a series of symbolic names formed from the common part and each of the integers between the first and the second integer, inclusive. As an example, <j0101>...<j0104> is interpreted as the symbolic names <j0101>, <j0102>, <j0103>, and <j0104>, in that order.

A character set mapping definition line shall exist for all symbolic names specified in Table 2-3, and shall define the coded character value that corresponds with the character glyph indicated in the table, or the coded character value that corresponds with the control character symbolic name. If the control characters commonly associated with the symbolic names in Table 2-4 are supported by the implementation, the symbolic name and the corresponding encoding value shall be included in the file. Additional unique symbolic names may be included. A coded character value can be represented by more than one symbolic name.

The encoding part shall be expressed as one (for single-byte character values) or more concatenated decimal, octal, or hexadecimal constants in the following formats:

"%cd%d", *<escape_char>* , *<decimal byte value>*

"%cx%x", *<escape_char>* , *<hexadecimal byte value>*

"%c%o", *<escape_char>* , *<octal byte value>*

Decimal constants shall be represented by two or three decimal digits, preceded by the escape character and the lowercase letter d; for example, \d05, \d97, or \d143. Hexadecimal constants shall be represented by two hexadecimal digits, preceded by the escape character and the lowercase letter x; for example, \x05, \x61, or \x8f. Octal constants shall be represented by two or three octal digits, preceded by the escape character; for example, \05, \141, or \217. In a portable charmap file, each constant shall represent an 8-bit byte. Implementations supporting other byte sizes may allow constants to represent values larger than those that can be represented in 8-bit bytes, and to allow additional digits in constants. When constants are concatenated for multibyte character values, they shall be of the same type, and interpreted in byte order from left to right. The manner in which constants are represented in the character is implementation defined. Omitting bytes from a multibyte character definition produces undefined results.

In lines defining ranges of symbolic names, the encoded value is the value for the first symbolic name in the range (the symbolic name preceding the ellipsis). Subsequent symbolic names defined by the range shall have encoding values in increasing order. For example, the line

```
<j0101>...<j0104>       \d129\d254
```

shall be interpreted as

```
<j0101>                 \d129\d254
<j0102>                 \d129\d255
<j0103>                 \d130\d0
<j0104>                 \d130\d1
```

The comment is optional.

For the interpretation of the dollar-sign and the number-sign, see 2.2.2.37 and 2.2.2.93.

2.4.2 Character Set Rationale. *(This subclause is not a part of P1003.2)*

The portable character set is listed in full so there is no dependency on the ISO/IEC 646 {1} (or historically ASCII) encoded character set, although the set is identical to the characters defined in the International Reference Version of ISO/IEC 646 {1}.

This standard poses no requirement that multiple character sets or code sets be supported, leaving this as a marketing differentiation for implementors. Although multiple *charmap* files are supported, it is the responsibility of the implementation to provide the file(s); if only one is provided, only that one will be accessible using the `localedef` utility's `−f` option (although in the case of just one file on the system, `−f` is not useful).

The statement about invariance in code sets for the portable character set is worded as it is to avoid precluding implementations where multiple incompatible code sets are available (say, ASCII and EBCDIC). The standard utilities cannot be expected to produce predictable results if they access portable characters that vary on the same implementation.

The character set description file provides:

- the capability to describe character set attributes (such as collation order or character classes) independent of character set encoding, and using only the characters in the portable character set. This makes it possible to create ''generic'' `localedef` source files for all code sets that share the portable character set (such as the ISO 8859 family or IBM Extended ASCII).

- standardized symbolic names for all characters in the portable character set, making it possible to refer to any such character regardless of encoding.

Implementations are free to describe more than one code set in a character set description file, as long as only one encoding exists for the characters in Table 2-3. For example, if an implementation defines ISO 8859-1 {5} as the primary code set, and ISO 8859-2 {6} as an alternate set, with each character from the alternate code set preceded in data by a shift code, a character set description file could contain a complete description of the primary set and those characters from the secondary that are not identical, the encoding of the latter including the shift code.

Implementations are free to choose their own symbolic names, as long as the names identified by this standard are also defined; this provides support for already existing ‘‘character names.’’

The names selected for the members of the portable character set follow the ISO 8859 {5} and the ISO/IEC 10646 {B11} standards. However, several commonly used UNIX system names occur as synonyms in the list:

— The traditional UNIX system names are used for control characters.

— The word ‘‘slash’’ is in addition to ‘‘solidus.’’

— The word ‘‘backslash’’ is in addition to ‘‘reverse-solidus.’’

— The word ‘‘hyphen’’ in addition to ‘‘hyphen-minus.’’

— The word ‘‘period’’ in addition to ‘‘full-stop.’’

— For the digits, the word ‘‘digit’’ is eliminated.

— For letters, the words ‘‘Latin Capital Letter’’ and ‘‘Latin Small Letter’’ are eliminated.

— The words ‘‘left-brace’’ and ‘‘right-brace’’ in addition to ‘‘left-curly-bracket’’ and ‘‘right-curly-bracket.’’

— The names of the digits are preferred over the numbers, to avoid possible confusion between ‘‘0’’ and ‘‘O’’, and between ‘‘1’’ and ‘‘l’’ (one and the letter ell).

The names for the control characters in Table 2-4 were taken from ISO 4873 {4}.

The charmap file was introduced to resolve problems with the portability of, especially, `localedef` sources. This standard assumes that the portable character set is constant across all locales, but does not prohibit implementations from supporting two incompatible codings, such as both ASCII and EBCDIC. Such ‘‘dual-support’’ implementations should have all charmaps and `localedef` sources encoded using one portable character set, in effect ‘‘cross-compiling’’ for the other environment. Naturally, charmaps (and `localedef` sources) are only portable without transformation between systems using the same encodings for the portable character set. They can, however, be transformed between two sets using only a subset of the actual characters (the portable set). However, the particular coded character set used for an application or an implementation does not necessarily imply different characteristics or collation: on the contrary, these attributes should in many cases be identical, regardless of code set. The charmap provides the capability to define a common locale definition for multiple code sets (the same `localedef` source can be used for code sets with different extended characters; the ability in the charmap to define ‘‘empty’’ names allows for characters missing in certain code sets).

In addition, several implementors have expressed an interest in using the charmap concept to provide the information required for support of multiple character sets. Examples of such information is encoding mechanism, string parsing rules, default font information, etc. Such extensions are not described here.

The <escape_char> declaration was added at the request of the international community to ease the creation of portable *charmap* files on terminals not implementing the default backslash escape. (This approach was adopted because this is a new interface invented by POSIX.2. Historical interfaces, such as the shell command language and awk, have not been modified to accommodate this type of terminal.) The <comment_char> declaration was added at the request of the international community to eliminate the potential confusion between the number sign and the pound sign.

The octal number notation with no leading zero required was selected to match 1
those of awk and tr and is consistent with that used by localedef. To avoid 1
confusion between an octal constant and the backreferences used in localedef 1
source, the octal, hexadecimal, and decimal constants must contain at least two 1
digits. As single-digit constants are relatively rare, this should not impose any 1
significant hardship. Each of the constants includes ‘‘two or more’’ digits to 1
account for systems in which the byte size is larger than eight bits. For example, 1
a Unicode system that has defined 16-bit bytes may require six octal, four hexade- 1
cimal, and five decimal digits. 1

The decimal notation is supported because some newer international standards define character values in decimal, rather than in the old column/row notation.

The charmap identifies the coded character sets supported by an implementation. At least one charmap must be provided, but no implementation is required to provide more than one. Likewise, implementations can allow users to generate new charmaps (for instance for a new version of the 8859 family of coded character sets), but does not have to do so. If users are allowed to create new charmaps, the system documentation must describe the rules that apply (for instance: ‘‘only coded character sets that are supersets of ISO/IEC 646 {1} IRV, no multibyte characters, etc.’’)

2.5 Locale

A *locale* is the definition of the subset of a user's environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behavior of components of the system. Category names correspond to the following environment variable names:

LC_CTYPE	Character classification and case conversion.
LC_COLLATE	Collation order.
LC_TIME	Date and time formats.
LC_NUMERIC	Numeric, nonmonetary formatting.
LC_MONETARY	Monetary formatting.
LC_MESSAGES	Formats of informative and diagnostic messages and interactive responses.

Conforming implementations shall provide the standard utilities and the interfaces in Annex B (if that option is supported) with the capability to modify their behavior based on the current locale, as defined in the Environment Variables subclause for each utility and interface.

Locales other than those supplied by the implementation can be created via the `localedef` utility (see 4.35), provided that the {POSIX2_LOCALEDEF} symbol is defined on the system; see 2.13.2. Otherwise, only the implementation-provided locale(s) can be used. The input to the utility is described in 2.5.2. The value that shall be used to specify a locale when using environment variables shall be the string specified as the *name* operand to the `localedef` utility when the locale was created. The strings `"C"` and `"POSIX"` are reserved as identifiers for the POSIX Locale (see 2.5.1.) When the value of a locale environment variable begins with a slash (`/`), it shall be interpreted as the pathname of the locale definition. If the value of the locale value does not begin with a slash, the mechanism used to locate the locale is implementation defined.

If different character sets are used by the locale categories, the results achieved by an application utilizing these categories is undefined. Likewise, if different code sets are used for the data being processed by interfaces whose behavior is dependent on the current locale, or the code set is different from the code set assumed when the locale was created, the result is also undefined.

2.5.0.1 Locale Rationale. *(This subclause is not a part of P1003.2)*

The description of locales is based on work performed in the UniForum Technical Committee Subcommittee on Internationalization. Wherever appropriate, keywords were taken from the C Standard {7} or the *X/Open Portability Guide* {B31}.

The value that shall be used to specify a locale when using environment variables is the name specified as the *name* operand to the `localedef` utility when the locale was created. This provides a verifiable method to create and invoke a locale.

The ‘‘object’’ definitions need not be portable, as long as ‘‘source’’ definitions are. Strictly speaking, ‘‘source’’ definitions are portable only between implementations using the same character set(s). Such ‘‘source’’ definitions can, if they use symbolic names only, easily be ported between systems using different code sets as long as the characters in the portable character set (Table 2-3) have common values between the code sets; this is frequently the case in historical implementations. Of course, this requires that the symbolic names used for characters outside the portable character set are identical between character sets. The definition of symbolic names for characters is outside the scope of this standard, but is certainly within the scope of other standards organizations. When such names are standardized, future versions of POSIX.2 should require the use of these names.

Applications can select the desired locale by invoking the *setlocale* () function (or equivalent) with the appropriate value. If the function is invoked with an empty string, the value of the corresponding environment variable is used. If the environment variable is unset or is set to the empty string, the implementation sets the appropriate environment as defined in 2.6.

2.5.1 POSIX Locale

Conforming implementations shall provide a *POSIX Locale* . The behavior of standard utilities in the POSIX Locale shall be as if the locale was defined via the `localedef` utility with input data from Table 2-5, Table 2-7, Table 2-9, Table 2-10, Table 2-8, and Table 2-11, all in 2.5.2.

The tables describe the characteristics and behavior of the POSIX Locale for data consisting entirely of characters from the portable character set in Table 2-3 and the control characters in Table 2-4. For characters other than those in the two tables, the behavior is unspecified.

The POSIX Locale can be specified by assigning the appropriate environment variables the values `"C"` or `"POSIX"`.

Table 2-5 shows the definition for the LC_CTYPE category.

Table 2-7 shows the definition for the LC_COLLATE category.

Table 2-8 shows the definition for the LC_MONETARY category.

Table 2-9 shows the definition for the LC_NUMERIC category.

Table 2-10 shows the definition for the LC_TIME category.

Table 2-11 shows the definition for the LC_MESSAGES category.

2.5.1.1 POSIX Locale Rationale. *(This subclause is not a part of P1003.2)*

The POSIX Locale is equal to the `"C"` locale, as specified in POSIX.1 {8}. To avoid being classified as a C-language function, the name has been changed to the *POSIX Locale*; the environment variable value can be either `"POSIX"`, or, for historical reasons, `"C"`.

The POSIX definitions mirror the historical UNIX system behavior.

The use of symbolic names for characters in the tables does not imply that the POSIX Locale must be described using symbolic character names, but merely that it may be advantageous to do so.

Implementations must define a locale as the ''default'' locale, to be invoked when no environment variables are set, or set to the empty string. This default locale can be the POSIX Locale or any other, implementation-defined locale. Some implementations may provide facilities for local installation administrators to set the default locale, customizing it for each location. This standard does not require such a facility.

2.5.2 Locale Definition

The capability to specify additional locales to those provided by an implementation is optional (see 2.13.2). If the option is not supported, only implementation-supplied locales are available. Such locales shall be documented using the format specified in this clause.

Locales can be described with the file format presented in this subclause. The file format is that accepted by the `localedef` utility (see 4.35). For the purposes of this subclause, the file is referred to as the *locale definition file*, but no locales shall be affected by this file unless it is processed by `localedef` or some similar mechanism. Any requirements in this subclause imposed upon ''the utility'' shall apply to `localedef` or to any other similar utility used to install locale information using the locale definition file format described here.

The locale definition file shall contain one or more locale category source definitions, and shall not contain more than one definition for the same locale category. If the file contains source definitions for more than one category, implementation-defined categories, if present, shall appear after the categories defined by this clause (2.5). A category source definition shall contain either the definition of a category or a `copy` directive. For a description of the `copy` directive, see 4.35. In the event that some of the information for a locale category, as specified in this standard, is missing from the locale source definition, the behavior of that category, if it is referenced, is unspecified.

A category source definition shall consist of a category header, a category body, and a category trailer. A category header shall consist of the character string naming of the category, beginning with the characters `LC_`. The category trailer shall consist of the string `END`, followed by one or more `<blank>`s and the string used in the corresponding category header.

The category body shall consist of one or more lines of text. Each line shall contain an identifier, optionally followed by one or more operands. Identifiers shall be either keywords, identifying a particular locale element, or collating elements. In addition to the keywords defined in this standard, the source can contain implementation-defined keywords. Each keyword within a locale shall have a unique name (i.e., two categories cannot have a commonly-named keyword); no keyword shall start with the characters `LC_`. Identifiers shall be separated from the operands by one or more `<blank>`s.

Operands shall be characters, collating elements, or strings of characters. Strings shall be enclosed in double-quotes. Literal double-quotes within strings shall be preceded by the *<escape character>*, described below. When a keyword is followed by more than one operand, the operands shall be separated by semicolons; `<blank>`s shall be allowed before and/or after a semicolon.

The first category header in the file can be preceded by a line modifying the comment character. It shall have the following format, starting in column 1:

    "comment_char %c\n", <comment character>

The comment character shall default to the number-sign (#). Blank lines and lines containing the *<comment char>* in the first position shall be ignored.

The first category header in the file can be preceded by a line modifying the escape character to be used in the file. It shall have the following format, starting in column 1:

    "escape_char %c\n", <escape character>

The escape character shall default to backslash, which is the character used in all examples shown in this standard.

A line can be continued by placing an escape character as the last character on the line; this continuation character shall be discarded from the input. Although the implementation need not accept any one portion of a continued line with a length exceeding {LINE_MAX} bytes, it shall place no limits on the accumulated length of the continued line. Comment lines shall not be continued on a subsequent line using an escaped `<newline>`.

Individual characters, characters in strings, and collating elements shall be represented using symbolic names, as defined below. In addition, characters can be represented using the characters themselves, or as octal, hexadecimal, or decimal constants. When nonsymbolic notation is used, the resultant locale definitions need not be portable between systems. The left angle bracket (<) is a reserved symbol, denoting the start of a symbolic name; when used to represent itself it shall be preceded by the escape character. The following rules apply to character representation:

(1) A character can be represented via a symbolic name, enclosed within angle brackets (< and >). The symbolic name, including the angle brackets, shall exactly match a symbolic name defined in the charmap file specified via the localedef −f option, and shall be replaced by a character value determined from the value associated with the symbolic name in the charmap file. The use of a symbolic name not found in the *charmap* file shall constitute an error, unless the category is LC_CTYPE or LC_COLLATE , in which case it shall constitute a warning condition (see localedef in 4.35 for a description of action resulting from errors and warnings). The specification of a symbolic name in a collating-element or collating-symbol clause that duplicates a symbolic name in the charmap file (if present) is an error. Use of the escape character or a right angle bracket within a symbolic name shall be invalid unless the character is preceded by the escape character.

Example : <c>;<c-cedilla> "<M><a><y>"

(2) A character can be represented by the character itself, in which case the value of the character is implementation defined. Within a string, the double-quote character, the escape character, and the right angle bracket character shall be escaped (preceded by the escape character) to be interpreted as the character itself. Outside strings, the characters

, ; < > *escape_char*

shall be escaped to be interpreted as the character itself.

Example : c β "May"

(3) A character can be represented as an octal constant. An octal constant shall be specified as the escape character followed by two or more octal digits. Each constant shall represent a byte value. Multibyte characters can be represented by concatenated constants.

Example : \143;\347;\143\150 "\115\141\171"

(4) A character can be represented as a hexadecimal constant. A hexadecimal constant shall be specified as the escape character followed by an x followed by two or more hexadecimal digits. Each constant shall represent a byte value. Multibyte characters can be represented by concatenated constants.

Example : \x63;\xe7;\x63\x68 "\x4d\x61\x79"

(5) A character can be represented as a decimal constant. A decimal constant shall be specified as the escape character followed by a d followed by two or more decimal digits. Each constant shall represent a byte value. Multibyte values can be represented by concatenated constants.

Example : \d99;\d231;\d99\d104 "\d77\d97\d121"

Implementations may accept single-digit octal, decimal, or hexadecimal constants following the escape character. Only characters existing in the character set for which the locale definition is created shall be specified, whether using symbolic

names, the characters themselves, or octal, decimal, or hexadecimal constants. If 2
a charmap file is present, only characters defined in the charmap can be specified 2
using octal, decimal, or hexadecimal constants. Symbolic names not present in 2
the charmap file can be specified and shall be ignored, as specified under item (1) 2
above. 2

2.5.2.0.1 Locale Definition Rationale. *(This subclause is not a part of P1003.2)*

The decision to separate the file format from the `localedef` utility description 1
was only partially editorial. Implementations may provide other interfaces than 1
`localedef`. Requirements on ''the utility,'' mostly concerning error messages, 1
are described in this way because they are meant to affect the other interfaces 1
implementations may provide as well as `localedef`. (This is similar to the philo- 1
sophy used by POSIX.1 {8} where the descriptions of the `tar` and `cpio` file formats 1
impose requirements on any utilities processing them.) 1

The text about {POSIX2_LOCALEDEF} does not mean that internationalization is optional; only that the functionality of the `localedef` utility is. Regular expressions, for instance, must still be able to recognize e.g., character class expressions such as `[[:alpha:]]`.

A possible analogy is with an applications development environment: while all conforming implementations must be capable of executing applications, not all need to have the development environment installed. The assumption is that the capability to modify the behavior of utilities (and applications) via locale settings must be supported. If the `localedef` utility is not present, then the only choice is to select an existing (presumably implementation-documented) locale. An implementation could, for example, chose to support only the POSIX Locale, which would in effect limit the amount of changes from historical implementations quite drastically. The `localedef` utility is still required, but would always terminate with an exit code indicating that no locale could be created. Supported locales must be documented using the syntax defined in 2.5. (This ensures that users can accurately determine what capabilities are provided. If the implementation decides to provide additional capabilities to the ones in 2.5, that is already provided for.)

If the option is present (i.e., locales can be created), then the `localedef` utility must be capable of creating locales based on the syntax and rules defined in 2.5. This does not mean that the implementation cannot also provide alternate means for creating locales.

The octal, decimal, and hexadecimal notations are the same employed by the 1
charmap facility (see 2.4.1). To avoid confusion between an octal constant and a 1
backreference, the octal, hexadecimal, and decimal constants must contain at 1
least two digits. As single-digit constants are relatively rare, this should not 1
impose any significant hardship. Each of the constants includes ''two or more'' 1
digits to account for systems in which the byte size is larger than eight bits. For 1
example, a Unicode system that has defined 16-bit bytes may require six octal, 1
four hexadecimal, and five decimal digits. 1

This standard is intended as an international (ISO/IEC) standard as well as an IEEE standard, and must therefore follow the ISO/IEC guidelines. One such rule is that characters outside the invariant part of ISO/IEC 646 {1} should not be used in portable specifications. The backslash character is not in the invariant part; the number-sign is, but with multiple representations: as a number-sign and as a pound sign. As far as general usage of these symbols, they are covered by the ''grandfather clause,'' but for newly defined interfaces, ISO has requested that POSIX provides alternate representations. Consequently, while the default escape character remains the backslash, and the default comment character is the number-sign, implementations are required to recognize alternative representations, identified in the applicable source file via the `escape_char` and `comment_char` keywords.

2.5.2.1 LC_CTYPE

The LC_CTYPE category shall define character classification, case conversion, and other character attributes. In addition, a series of characters can be represented by three adjacent periods representing an ellipsis symbol (''. . .''). The ellipsis specification shall be interpreted as meaning that all values between the values preceding and following it represent valid characters. The ellipsis specification only shall be valid within a single encoded character set. An ellipsis shall be interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

Example: `\x30;...;\x39;` includes in the character class all characters with encoded values between the endpoints.

The following keywords shall be recognized. In the descriptions, the term ''automatically included'' means that it shall not be an error to either include the referenced characters or to omit them; the implementation shall provide them if missing and accept them silently if present.

`copy` Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.

`upper` Define characters to be classified as uppercase letters. No character specified for the keywords `cntrl`, `digit`, `punct`, or `space` shall be specified. If this keyword is not specified, the uppercase letters `A` through `Z`, as defined in Table 2-3 (see 2.4.1), shall automatically belong to this class, with implementation-defined character values.

`lower` Define characters to be classified as lowercase letters. No character specified for the keywords `cntrl`, `digit`, `punct`, or `space` shall be specified. If this keyword is not specified, the lowercase letters `a` through `z`, as defined in Table 2-3 (see 2.4.1), shall automatically belong to this class, with implementation-defined character values.

Table 2-5 – LC_CTYPE Category Definition in the POSIX Locale

```
LC_CTYPE
# The following is the POSIX Locale LC_CTYPE.
# "alpha" is by default "upper" and "lower"
# "alnum" is by definition "alpha" and "digit"
# "print" is by default "alnum", "punct" and the <space> character
# "graph" is by default "alnum" and "punct"

#
upper   <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>
#
lower   <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>
#
digit   <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;<eight>;<nine>
#
space   <tab>;<newline>;<vertical-tab>;<form-feed>;<carriage-return>;<space>
#
cntrl   <alert>;<backspace>;<tab>;<newline>;<vertical-tab>;\
        <form-feed>;<carriage-return>;\
        <NUL>;<SOH>;<STX>;<ETX>;<EOT>;<ENQ>;<ACK>;<SO>;\
        <SI>;<DLE>;<DC1>;<DC2>;<DC3>;<DC4>;<NAK>;<SYN>;\
        <ETB>;<CAN>;<EM>;<SUB>;<ESC>;<IS4>;<IS3>;<IS2>;\
        <IS1>;<DEL>
#
punct   <exclamation-mark>;<quotation-mark>;<number-sign>;\
        <dollar-sign>;<percent-sign>;<ampersand>;<apostrophe>;\
        <left-parenthesis>;<right-parenthesis>;<asterisk>;\
        <plus-sign>;<comma>;<hyphen>;<period>;<slash>;\
        <colon>;<semicolon>;<less-than-sign>;<equals-sign>;\
        <greater-than-sign>;<question-mark>;<commercial-at>
        <left-square-bracket>;<backslash>;<right-square-bracket>;\
        <circumflex>;<underline>;<grave-accent>;\
        <left-curly-bracket>;<vertical-line>;<right-curly-bracket>;<tilde>
#
xdigit  <zero>;<one>;<two>;<three>;<four>;<five>;<six>;<seven>;<eight>;\
        <nine>;<A>;<B>;<C>;<D>;<E>;<F>;<a>;<b>;<c>;<d>;<e>;<f>
#
blank   <space>;<tab>
#
toupper (<a>,<A>);(<b>,<B>);(<c>,<C>);(<d>,<D>);(<e>,<E>);\
        (<f>,<F>);(<g>,<G>);(<h>,<H>);(<i>,<I>);(<j>,<J>);\
        (<k>,<K>);(<l>,<L>);(<m>,<M>);(<n>,<N>);(<o>,<O>);\
        (<p>,<P>);(<q>,<Q>);(<r>,<R>);(<s>,<S>);(<t>,<T>);\
        (<u>,<U>);(<v>,<V>);(<w>,<W>);(<x>,<X>);(<y>,<Y>);(<z>,<Z>)
#
tolower (<A>,<a>);(<B>,<b>);(<C>,<c>);(<D>,<d>);(<E>,<e>);\
        (<F>,<f>);(<G>,<g>);(<H>,<h>);(<I>,<i>);(<J>,<j>);\
        (<K>,<k>);(<L>,<l>);(<M>,<m>);(<N>,<n>);(<O>,<o>);\
        (<P>,<p>);(<Q>,<q>);(<R>,<r>);(<S>,<s>);(<T>,<t>);\
        (<U>,<u>);(<V>,<v>);(<W>,<w>);(<X>,<x>);(<Y>,<y>);(<Z>,<z>)
END LC_CTYPE
```

`alpha` Define characters to be classified as letters. No character specified for the keywords `cntrl`, `digit`, `punct`, or `space` shall be specified. In addition, characters classified as either `upper` or `lower` shall automatically belong to this class.

`digit` Define the characters to be classified as numeric digits. Only the digits `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, and `9` shall be specified, and in ascending sequence by numerical value. If this keyword is not specified, the digits `0` through `9`, as defined in Table 2-3 (see 2.4.1), shall automatically belong to this class, with implementation-defined character values.

`space` Define characters to be classified as white-space characters. No character specified for the keywords `upper`, `lower`, `alpha`, `digit`, `graph`, or `xdigit` shall be specified. If this keyword is not specified, the characters `<space>`, `<form-feed>`, `<newline>`, `<carriage-return>`, `<tab>`, and `<vertical-tab>`, as defined in Table 2-3 (see 2.4.1), shall automatically belong to this class, with implementation-defined character values. Any characters included in the class `blank` shall be automatically included.

`cntrl` Define characters to be classified as control characters. No character specified for the keywords `upper`, `lower`, `alpha`, `digit`, `punct`, `graph`, `print`, or `xdigit` shall be specified.

`punct` Define characters to be classified as punctuation characters. No character specified for the keywords `upper`, `lower`, `alpha`, `digit`, `cntrl`, `xdigit`, or as the `<space>` character shall be specified.

`graph` Define characters to be classified as printable characters, not including the `<space>` character. If this keyword is not specified, characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, and `punct` shall belong to this character class. No character specified for the keyword `cntrl` shall be specified.

`print` Define characters to be classified as printable characters, including the `<space>` character. If this keyword is not provided, characters specified for the keywords `upper`, `lower`, `alpha`, `digit`, `xdigit`, `punct`, and the `<space>` character shall belong to this character class. No character specified for the keyword `cntrl` shall be specified.

`xdigit` Define the characters to be classified as hexadecimal digits. Only the characters defined for the class `digit` shall be specified, in ascending sequence by numerical value, followed by one or more sets of six characters representing the hexadecimal digits 10 through 15, with each set in ascending order (for example `A`, `B`, `C`, `D`, `E`, `F`, `a`, `b`, `c`, `d`, `e`, `f`). If this keyword is not specified, the digits `0` through `9`, the uppercase letters `A` through

F, and the lowercase letters a through f, as defined in Table 2-3 (see 2.4.1), shall automatically belong to this class, with implementation-defined character values.

`blank` Define characters to be classified as <blank> characters. If this keyword is unspecified, the characters <space> and <tab> shall belong to this character class.

`toupper` Define the mapping of lowercase letters to uppercase letters. The operand shall consist of character pairs, separated by semicolons. The characters in each character pair shall be separated by a comma and the pair enclosed by parentheses. The first character in each pair shall be the lowercase letter, the second the corresponding uppercase letter. Only characters specified for the keywords `lower` and `upper` shall be specified. If this keyword is not specified, the lowercase letters a through z, and their corresponding uppercase letters A through Z, as defined in Table 2-3 (see 2.4.1), shall automatically be included, with implementation-defined character values.

`tolower` Define the mapping of uppercase letters to lowercase letters. The operand shall consist of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair enclosed by parentheses. The first character in each pair shall be the uppercase letter, the second the corresponding lowercase letter. Only characters specified for the keywords `lower` and `upper` shall be specified.

The `tolower` keyword is optional. If specified, the uppercase letters A through Z, as defined in Table 2-3, and their corresponding lowercase letter, shall be specified. If this keyword is not specified, the mapping shall be the reverse mapping of the one specified for `toupper`.

Table 2-6 shows the allowed character class combinations.

Table 2-6 – Valid Character Class Combinations

In	Can Also Belong To										
Class	upper	lower	alpha	digit	space	cntrl	punct	graph	print	xdigit	blank	
upper	-	-	M	X	X	X	X	D	D	-	X	
lower	-	-	M	X	X	X	X	D	D	-	X	
alpha	-	-	-	X	X	X	X	D	D	-	X	
digit	X	X	X	-	X	X	X	D	D	-	X	
space	X	X	X	X	-	-	*	*	*	X	-	2
cntrl	X	X	X	X	-	-	X	X	X	X	-	2
punct	X	X	X	X	-	X	-	D	D	X	-	
graph	-	-	-	-	-	X	-	-	-	-	-	
print	-	-	-	-	-	X	-	-	-	-	-	
xdigit	-	-	-	-	X	X	X	D	D	-	X	
blank	X	X	X	X	M	-	*	*	*	X	-	2

NOTES:

(1) Explanation of codes:

M Always

D Default; belongs to class if not specified

\- Permitted

X Mutually exclusive

* See note (2)

(2) The `<space>` character, which is part of the `space` and `blank` classes, cannot belong to `punct` or `graph`, but automatically shall belong to the `print` class. Other `space` or `blank` characters can be classified as `punct`, `graph`, and/or `print`.

2.5.2.1.1 LC_CTYPE Rationale. *(This subclause is not a part of P1003.2)*

The LC_CTYPE category primarily is used to define the encoding-independent aspects of a character set, such as character classification. In addition, certain encoding-dependent characteristics are also defined for an application via the LC_CTYPE category. POSIX.2 does not mandate that the encoding used in the locale is the same as the one used by the application, because an implementation may decide that it is advantageous to define locales in a system-wide encoding rather than having multiple, logically identical locales in different encodings, and to convert from the application encoding to the system-wide encoding on usage. Other implementations could require encoding-dependent locales.

In either case, the LC_CTYPE attributes that are directly dependent on the encoding, such as `mb_cur_max` and the display width of characters, are not user-specifiable in a locale source, and are consequently not defined as keywords.

As the LC_CTYPE character classes are based on the C Standard {7} character-class definition, the category does not support multicharacter elements. For instance, the German character <sharp-s> is traditionally classified as a lowercase letter. There is no corresponding uppercase letter; in proper capitalization of German text the <sharp-s> will be replaced by SS; i.e., by two characters. This kind of conversion is outside the scope of the toupper and tolower keywords.

Where POSIX.2 specifies that only certain characters can be specified, as for the 1
keywords digit and xdigit, the specified characters must be from the portable 1
character set, as shown. As an example, only the Arabic digits 0 through 9 are 1
acceptable as digits. 1

The character classes digit, xdigit, lower, upper, and space have a set of 2
automatically included characters. These only need to be specified if the charac- 2
ter values (i.e., encoding) differs from the implementation default values. 2

The definition of character class digit requires that only ten characters—the 2
ones defining digits—can be specified; alternate digits (e.g., Hindi or Kanji) can- 2
not be specified here. However, the encoding may vary if an implementation sup- 2
ports more than one encoding. 2

The definition of character class xdigit requires that the characters included in 2
character class digit are included here also, and allows for different symbols for 2
the hexadecimal digits 10 through 15. 2

2.5.2.2 LC_COLLATE

A collation sequence definition shall define the relative order between collating elements (characters and multicharacter collating elements) in the locale. This order is expressed in terms of collation values; i.e., by assigning each element one or more collation values (also known as collation weights). This does not imply that implementations shall assign such values, but that ordering of strings using the resultant collation definition in the locale shall behave as if such assignment is done and used in the collation process. The collation sequence definition shall be used by regular expressions, pattern matching, and sorting. The following capabilities are provided:

(1) **Multicharacter collating elements**. Specification of multicharacter collating elements (i.e., sequences of two or more characters to be collated as an entity).

(2) **User-defined ordering of collating elements**. Each collating element shall be assigned a collation value defining its order in the character (or basic) collation sequence. This ordering is used by regular expressions and pattern matching and, unless collation weights are explicitly specified, also as the collation weight to be used in sorting.

(3) **Multiple weights and equivalence classes**. Collating elements can be assigned one or more (up to the limit {COLL_WEIGHTS_MAX}) collating weights for use in sorting. The first weight is hereafter referred to as the primary weight.

(4) **One-to-Many mapping**. A single character is mapped into a string of collating elements.

(5) **Many-to-Many substitution**. A string of one or more characters is substituted by another string (or an empty string, i.e., the character or characters shall be ignored for collation purposes).

(6) **Equivalence class definition**. Two or more collating elements have the same collation value (primary weight).

(7) **Ordering by weights**. When two strings are compared to determine their relative order, the two strings are first broken up into a series of collating elements, and each successive pair of elements are compared according to the relative primary weights for the elements. If equal, and more than one weight has been assigned, then the pairs of collating elements are recompared according to the relative subsequent weights, until either a pair of collating elements compare unequal or the weights are exhausted.

The following keywords shall be recognized in a collation sequence definition. They are described in detail in the following subclauses.

`copy`	Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.
`collating-element`	Define a collating-element symbol representing a multicharacter collating element. This keyword is optional.
`collating-symbol`	Define a collating symbol for use in collation order statements. This keyword is optional.
`order_start`	Define collation rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.
`order_end`	Specify the end of the collation-order statements.

2.5.2.2.1 `collating-element` Keyword

In addition to the collating elements in the character set, the `collating-element` keyword shall be used to define multicharacter collating elements. The syntax is

`"collating-element %s from %s\n"`, *<collating-symbol>* , *<string>*

The *<collating-symbol>* operand shall be a symbolic name, enclosed between angle brackets (< and >), and shall not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. The string operand shall be a string of two or more characters that shall collate as an entity. A *<collating-element>* defined via this keyword is only recognized with the LC_COLLATE category.

Table 2-7 – LC_COLLATE Category Definition in the POSIX Locale

```
LC_COLLATE
# This is the POSIX Locale definition for the LC_COLLATE category.
# The order is the same as in the ASCII code set.
order_start forward
<NUL>
<SOH>
<STX>
<ETX>
<EOT>
<ENQ>
<ACK>
<alert>
<backspace>
<tab>
<newline>
<vertical-tab>
<form-feed>
<carriage-return>
<SO>
<SI>
<DLE>
<DC1>
<DC2>
<DC3>
<DC4>
<NAK>
<SYN>
<ETB>
<CAN>
<EM>
<SUB>
<ESC>
<IS4>
<IS3>
<IS2>
<IS1>
<space>
<exclamation-mark>
<quotation-mark>
<number-sign>
<dollar-sign>
<percent-sign>
<ampersand>
<apostrophe>
<left-parenthesis>
<right-parenthesis>
<asterisk>
```

Table 2-7 – LC_COLLATE Category Definition in the POSIX Locale (*continued*)

<plus-sign>
<comma>
<hyphen>
<period>
<slash>
<zero>
<one>
<two>
<three>
<four>
<five>
<six>
<seven>
<eight>
<nine>
<colon>
<semicolon>
<less-than-sign>
<equals-sign>
<greater-than-sign>
<question-mark>
<commercial-at>
<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>

Table 2-7 – LC_COLLATE Category Definition in the POSIX Locale (*concluded*)

```
<left-square-bracket>
<backslash>
<right-square-bracket>
<circumflex>
<underline>
<grave-accent>
<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>
<left-curly-bracket>
<vertical-line>
<right-curly-bracket>
<tilde>
<DEL>
order_end
#
END LC_COLLATE
```

Example:

```
      collating-element <ch> from <c><h>
      collating-element <e-acute> from <acute><e>
      collating-element <ll> from ll
```

2.5.2.2.2 `collating-symbol` Keyword

This keyword shall be used to define symbols for use in collation sequence statements; i.e., between the `order_start` and the `order_end` keywords. The syntax is

```
"collating-symbol %s\n", <collating-symbol>
```

The *<collating-symbol>* shall be a symbolic name, enclosed between angle brackets (`<` and `>`), and shall not duplicate any symbolic name in the current charmap file (if any), or any other symbolic name defined in this collation definition. A *<collating-symbol>* defined via this keyword is only recognized with the LC_COLLATE category. 1

Example:

```
collating-symbol <UPPER_CASE>
collating-symbol <HIGH>
```

2

2.5.2.2.3 `order_start` Keyword

The `order_start` keyword shall precede collation order entries and also defines the number of weights for this collation sequence definition and other collation rules.

The syntax of the `order_start` keyword is:

```
"order_start %s;%s;...;%s\n", <sort-rules> , <sort-rules> ...
```

The operands to the `order_start` keyword are optional. If present, the operands define rules to be applied when strings are compared. The number of operands define how many weights each element is assigned; if no operands are present, one `forward` operand is assumed. If present, the first operand defines rules to be applied when comparing strings using the first (primary) weight; the second when comparing strings using the second weight, and so on. Operands shall be separated by semicolons (`;`). Each operand shall consist of one or more collation directives, separated by commas (`,`). If the number or operands exceeds the {COLL_WEIGHTS_MAX} limit, the utility shall issue a warning message. The following directives shall be supported:

`forward` Specifies that comparison operations for the weight level shall proceed from start of string towards the end of string.

`backward` Specifies that comparison operations for the weight level shall proceed from end of string towards the beginning of string.

2

`position` Specifies that comparison operations for the weight level will consider the relative position of non-`IGNORE`d ele- 2
ments in the strings. The string containing a non- 2

> IGNOREd element after the fewest IGNOREd collating elements from the start of the compare shall collate first. If both strings contain a non-IGNOREd character in the same relative position, the collating values assigned to the elements shall determine the ordering. In case of equality, subsequent non-IGNOREd characters shall be considered in the same manner.

The directives `forward` and `backward` are mutually exclusive.

Example :

```
order_start     forward;backward
```

If no operands are specified, a single `forward` operand shall be assumed.

2.5.2.2.4 Collation Order

The `order_start` keyword shall be followed by collating element entries. The syntax for the collating element entries is

`"%s %s;%s;...;%s\n"`, *<collating-element>* , *<weight>* , *<weight>* , . . .

Each *collating-element* shall consist of either a character (in any of the forms defined in 2.5.2), a *<collating-element>* , a *<collating-symbol>* , an ellipsis, or the special symbol UNDEFINED. The order in which collating elements are specified determines the character collation sequence, such that each collating element shall compare less than the elements following it. The NUL character shall compare lower than any other character.

A *<collating-element>* shall be used to specify multicharacter collating elements, and indicates that the character sequence specified via the *<collating-element>* is to be collated as a unit and in the relative order specified by its place.

A *<collating-symbol>* shall be used to define a position in the relative order for use in weights.

The ellipsis symbol (‘‘`...`’’) specifies that a sequence of characters shall collate according to their encoded character values. It shall be interpreted as indicating that all characters with a coded character set value higher than the value of the character in the preceding line, and lower than the coded character set value for the character in the following line, in the current coded character set, shall be placed in the character collation order between the previous and the following character in ascending order according to their coded character set values. An initial ellipsis shall be interpreted as if the preceding line specified the NUL character, and a trailing ellipsis as if the following line specified the highest coded character set value in the current coded character set. An ellipsis shall be treated as invalid if the preceding or following lines do not specify characters in the current coded character set. The use of the ellipsis symbol ties the definition to a specific coded character set and may preclude the definition from being portable between implementations.

The symbol UNDEFINED shall be interpreted as including all coded character set values not specified explicitly or via the ellipsis symbol. Such characters shall be inserted in the character collation order at the point indicated by the symbol, and in ascending order according to their coded character set values. If no `UNDEFINED` symbol is specified, and the current coded character set contains characters not specified in this clause, the utility shall issue a warning message and place such characters at the end of the character collation order.

The optional operands for each collation-element shall be used to define the primary, secondary, or subsequent weights for the collating element. The first operand specifies the relative primary weight, the second the relative secondary weight, and so on. Two or more collation-elements can be assigned the same weight; they belong to the same *equivalence class* if they have the same primary weight. Collation shall behave as if, for each weight level, `IGNORE`d elements are removed. Then each successive pair of elements shall be compared according to the relative weights for the elements. If the two strings compare equal, the process shall be repeated for the next weight level, up to the limit {COLL_WEIGHTS_MAX}.

Weights shall be expressed as characters (in any of the forms specified in 2.5.2), *<collating-symbol>* s, *<collating-element>* s, an ellipsis, or the special symbol `IGNORE`. A single character, a *<collating-symbol>* , or a *<collating-element>* shall represent the relative order in the character collating sequence of the character or symbol, rather than the character or characters themselves.

One-to-many mapping is indicated by specifying two or more concatenated characters or symbolic names. Thus, if the character ‘‘`<eszet>`’’ is given the string `<s><s>` as a weight, comparisons shall be performed as if all occurrences of the character `<eszet>` are replaced by `<s><s>`. If it is desirable to define `<eszet>` and `<s><s>` as an equivalence class, then a collating-element must be defined for the string ‘‘ss’’,as in the example below.

All characters specified via an ellipsis shall by default be assigned unique weights, equal to the relative order of characters. Characters specified via an explicit or implicit `UNDEFINED` special symbol shall by default be assigned the same primary weight (i.e., belong to the same equivalence class). An ellipsis symbol as a weight shall be interpreted to mean that each character in the sequence shall have unique weights, equal to the relative order of their character in the character collation sequence. Secondary and subsequent weights have unique values. The use of the ellipsis as a weight shall be treated as an error if the collating element is neither an ellipsis nor the special symbol `UNDEFINED`.

The special keyword `IGNORE` as a weight shall indicate that when strings are compared using the weights at the level where `IGNORE` is specified, the collating element shall be ignored; i.e., as if the string did not contain the collating element. In regular expressions and pattern matching, all characters that are `IGNORE`d in their primary weight form an equivalence class.

An empty operand shall be interpreted as the collating-element itself.

For example, the order statement

```
<a>      <a>;<a>
```

is equal to

```
<a>
```

An ellipsis can be used as an operand if the collating-element was an ellipsis, and shall be interpreted as the value of each character defined by the ellipsis.

The collation order as defined in this clause defines the interpretation of bracket 1
expressions in regular expressions (see 2.8.3.2). 1

Example :

```
order_start   forward;backward
UNDEFINED     IGNORE;IGNORE
<LOW>
<space>       <LOW>;<space>
...           <LOW>;...
<a>           <a>;<a>
<a-acute>     <a>;<a-acute>
<a-grave>     <a>;<a-grave>
<A>           <a>;<A>
<A-acute>     <a>;<A-acute>
<A-grave>     <a>;<A-grave>
<ch>          <ch>;<ch>
<Ch>          <ch>;<Ch>
<s>           <s>;<s>
```
2
```
<eszet>       <s><s>;<eszet><eszet>
...           <HIGH>;...
<HIGH>
order_end
```

This example is interpreted as follows:

(1) The `UNDEFINED` means that all characters not specified in this definition (explicitly or via the ellipsis) shall be ignored for collation purposes; for regular expression purposes they are ordered first.

(2) All characters between `<space>` and `<a>` shall have the same primary equivalence class and individual secondary weights based on their ordinal encoded values.

(3) All characters based on the upper- or lowercase character `a` belong to the same primary equivalence class.

(4) The multicharacter collating element `<c><h>` is represented by the collating symbol `<ch>` and belongs to the same primary equivalence class as the multicharacter collating element `<C><h>`.

(5) Note that it is not possible to use the collating element `<ss>` as a weight 1
and expect it to be expanded to the string ''ss''. When used as a weight, 1
any collating-element represents the relative order assigned to it in the 1
character collation sequence, not the string from which it was derived 1
(compare with `<ch>`). 1

2.5.2.2.5 `order_end` Keyword

The collating order entries shall be terminated with an `order_end` keyword.

2.5.2.2.6 LC_COLLATE Rationale. *(This subclause is not a part of P1003.2)*

The LC_COLLATE category governs the collation order in the locale, and thus the processing of the C Standard {7} *strxfrm* () and *strcoll* () functions, as well as a number of POSIX. 2 utilities.

The rules governing collation depends to some extent on the use. At least five different levels of increasingly complex collation rules can be distinguished:

(1) Byte/machine code order. This is the historical collation order in the UNIX system and many proprietary operating systems. Collation is here done character by character, without any regard to context. The primary virtue is that it usually is quite fast, and also completely deterministic; it works well when the native machine collation sequence matches the user expectations.

(2) Character order. On this level, collation is also done character by character, without regard to context. The order between characters is, however, not determined by the code values, but on the user's expectations of the ''correct'' order between characters. In addition, such a (simple) collation order can specify that certain characters collate equal (e.g., upper- and lowercase letters).

(3) String ordering. On this level, entire strings are compared based on relatively straightforward rules. At this level, several ''passes'' may be required to determine the order between two strings. Characters may be ignored in some passes, but not in others; the strings may be compared in different directions; and simple string substitutions may be made before strings are compared. This level is best described as ''dictionary'' ordering; it is based on the spelling, not the pronunciation, or meaning, of the words.

(4) Text search ordering. This is a further refinement of the previous level, best described as ''telephone book ordering''; some common homonyms (words spelled differently but with same pronunciation) are collated together; numbers are collated as if spelled with words, and so on.

(5) Semantic level ordering. Words and strings are collated based on their meaning; entire words (such as ''the'') are eliminated, the ordering is not deterministic. This usually requires special software, and is highly dependent on the intended use.

While the historical collation order formally is at level 1, for the English language it corresponds roughly to elements at level 2. The user expects to see the output from the `ls` utility sorted very much as as it would be in a dictionary. While telephone book ordering would be an optimal goal for standard collation, this was ruled out as the order would be language dependent. Furthermore, a requirement was that the order must be determined solely from the text string and the collation rules; no external information (e.g., ‘‘pronunciation dictionaries’’) could be required.

As a result, the goal for the collation support is at level 3. This also matches the requirements for the proposed Canadian collation order, as well as other, known collation requirements for alphabetic scripts. It specifically rules out collation based on pronunciation rules, or based on semantic analysis of the text.

The syntax for the LC_COLLATE category source is the result of a cooperative effort between representatives for many countries and organizations working with international issues, such as UniForum, X/Open, and ISO, and it meets the requirements for level 3, and has been verified to produce the correct result with examples based on French, Canadian, and Danish collation order, as well as meeting the requirements in the X/Open Portability Guide, Issue 3. {B31}. Because it supports multicharacter collating elements, it is also capable of supporting collation in code sets where a character is expressed using nonspacing characters followed by the base character (such as ISO 6937 {B6}).

The directives that can be specified in an operand to the `order_start` keyword are based on the requirements specified in several proposed standards and in customary use. The following is a rephrasing of rules defined for ‘‘lexical ordering in English and French’’ by the Canadian Standards Association (text is brackets is rephrased):

(1) Once special characters ([punctuation]) have been removed from original strings, the ordering is determined by scanning forward (left to right) [disregarding case and diacriticals].

(2) In case of equivalence, special characters are once again removed from original strings and the ordering is determined scanning backward (starting from the rightmost character of the string and back), character by character, [disregarding case but considering diacriticals].

(3) In case of repeated equivalence, special characters are removed again from original strings and the ordering is determined scanning forward, character by character, [considering both case and diacriticals].

(4) If there is still an ordering equivalence after rules (1) through (3) have been applied, then only special characters and the position they occupy in the string are considered to determine ordering. The string that has a special character in the lowest position comes first. If two strings have a special character in the same position, the character [with the lowest collation value] comes first. In case of equality, the other special characters are considered until there is a difference or all special characters have been exhausted.

It is estimated that the standard covers the requirements for all European languages, and no particular problems are anticipated with Slavic or Middle East character sets.

The Far East (particularly Japanese/Chinese) collations are often based on contextual information and pronunciation rules (the same ideogram can have different meanings and different pronunciations). Such collation, in general, falls outside the desired goal of the standard. There are, however, several other collation rules (stroke/radical, or ‘‘most common pronunciation’’) which can be supported with the mechanism described here.

Previous drafts contained a `substitute` statement, which performed a regular expression style replacement before string compares. It has been withdrawn based on balloter objections that it was not required for the types of ordering POSIX. 2 is aimed at.

The character (and collating element) order is defined by the order in which characters and elements are specified between the `order_start` and `order_end` keywords. This character order is used in range expressions in regular expressions (see 2.8). Weights assigned to the characters and elements defines the collation sequence; in the absence of weights, the character order is also the collation sequence.

The `position` keyword was introduced to provide the capability to consider, in a compare, the relative position of non-`IGNORE`d characters. As an example, consider the two strings ‘‘o-ring’’ and ‘‘or-ing’’. Assuming the hyphen is `IGNORE`d on the first pass, the two strings will compare equal, and the position of the hyphen is immaterial. On second pass, all characters except the hyphen are `IGNORE`d, and in the normal case the two strings would again compare equal. By taking position into account, the first collates before the second.

2.5.2.3 LC_MONETARY

The LC_MONETARY category shall define the rules and symbols that shall be used to format monetary numeric information. The operands are strings. For some keywords, the strings can contain only integers. Keywords that are not provided, string values set to the empty string (`""`), or integer keywords set to −1, shall be used to indicate that the value is unspecified. The following keywords shall be recognized:

`copy` Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.

`int_curr_symbol` The international currency symbol. The operand shall be a four-character string, with the first three characters containing the alphabetic international currency symbol in accordance with those specified in ISO 4217 {3} (*Codes for the representation of currencies and funds*). The fourth character shall be the character used to separate the international currency symbol from the monetary quantity.

`currency_symbol` The string that shall be used as the local currency symbol.

`mon_decimal_point` The operand is a string containing the symbol that shall be used as the decimal delimiter in monetary formatted quantities. In contexts where other standards

Table 2-8 – LC_MONETARY Category Definition in the POSIX Locale

```
LC_MONETARY
# This is the POSIX Locale definition for
# the LC_MONETARY category.
#
int_curr_symbol     ""
currency_symbol     ""
mon_decimal_point   ""
mon_thousands_sep   ""
mon_grouping        ""
positive_sign       ""
negative_sign       ""
int_frac_digits     -1
p_cs_precedes       -1
p_sep_by_space      -1
n_cs_precedes       -1
n_sep_by_space      -1
p_sign_posn         -1
n_sign_posn         -1
#
END LC_MONETARY
```

limit the `mon_decimal_point` to a single byte, the result of specifying a multibyte operand is unspecified.

`mon_thousands_sep` The operand is a string containing the symbol that shall be used as a separator for groups of digits to the left of the decimal delimiter in formatted monetary quantities. In contexts where other standards limit the `mon_thousands_sep` to a single byte, the result of specifying a multibyte operand is unspecified.

`mon_grouping` Define the size of each group of digits in formatted monetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the preceding groups. If the last integer is not −1, then the size of the previous group (if any) shall be repeatedly used for the remainder of the digits. If the last integer is −1, then no further grouping shall be performed.

`positive_sign` A string that shall be used to indicate a nonnegative-valued formatted monetary quantity.

`negative_sign` A string that shall be used to indicate a negative-valued formatted monetary quantity.

`int_frac_digits` An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using `int_curr_symbol`.

`frac_digits` An integer representing the number of fractional digits (those to the right of the decimal delimiter) to be written in a formatted monetary quantity using `currency_symbol`.

`p_cs_precedes` An integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a nonnegative formatted monetary quantity, and set to 0 if the symbol succeeds the value.

`p_sep_by_space` An integer set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a nonnegative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

`n_cs_precedes` An integer set to 1 if the `currency_symbol` or `int_curr_symbol` precedes the value for a negative formatted monetary quantity, and set to 0 if the symbol succeeds the value.

`n_sep_by_space` An integer set to 0 if no space separates the `currency_symbol` or `int_curr_symbol` from the value for a negative formatted monetary quantity, set to 1 if a space separates the symbol from the value, and set to 2 if a space separates the symbol and the sign string, if adjacent.

`p_sign_posn` An integer set to a value indicating the positioning of the `positive_sign` for a nonnegative formatted monetary quantity. The following integer values shall be recognized:

- `0` Parentheses enclose the quantity and the `currency_symbol` or `int_curr_symbol`.
- `1` The sign string precedes the quantity and the `currency_symbol` or `int_curr_symbol`.
- `2` The sign string succeeds the quantity and the `currency_symbol` or `int_curr_symbol`.
- `3` The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.
- `4` The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

`n_sign_posn` An integer set to a value indicating the positioning of the `negative_sign` for a negative formatted mone- 1
tary quantity. The following integer values shall be recognized:

`0` Parentheses enclose the quantity and the `currency_symbol` or `int_curr_symbol`.

`1` The sign string precedes the quantity and the `currency_symbol` or `int_curr_symbol`.

`2` The sign string succeeds the quantity and the `currency_symbol` or `int_curr_symbol`.

`3` The sign string immediately precedes the `currency_symbol` or `int_curr_symbol`.

`4` The sign string immediately succeeds the `currency_symbol` or `int_curr_symbol`.

2.5.2.3.1 LC_MONETARY Rationale. *(This subclause is not a part of P1003.2)*

The currency symbol does not appear in LC_MONETARY because it is not defined in the C Standard's {7} C locale.

The C Standard {7} limits the size of decimal points and thousands delimiters to 2
single-byte values. In locales based on multibyte coded character sets this cannot 2
be enforced, obviously; this standard does not prohibit such characters, but makes 2
the behavior unspecified [in the text ''In contexts where other standards . . .'']. 2

The grouping specification is based on, but not identical to, the C Standard {7}. 2
The ''–1'' signals that no further grouping shall be performed, the equivalent of 2
{CHAR_MAX} in the C Standard {7}). 2

The locale definition is an extension of the C Standard {7} *localeconv*() specification. In particular, rules on how `currency_symbol` is treated are extended to also cover `int_curr_symbol`, and `p_set_by_space` and `n_sep_by_space` have been augmented with the value 2, which places a space between the sign and the symbol (if they are adjacent; otherwise it should be treated as a 0). The following table shows the result of various combinations:

		p_sep_by_space 2	1	0
p_cs_precedes = 1	p_sign_posn = 0	($1.25)	($ 1.25)	($1.25)
	p_sign_posn = 1	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 2	$1.25 +	$ 1.25+	$1.25+
	p_sign_posn = 3	+ $1.25	+$ 1.25	+$1.25
	p_sign_posn = 4	$ +1.25	$+ 1.25	$+1.25
p_cs_precedes = 0	p_sign_posn = 0	(1.25 $)	(1.25 $)	(1.25$)
	p_sign_posn = 1	+1.25 $	+1.25 $	+1.25$
	p_sign_posn = 2	1.25$ +	1.25 $+	1.25$+
	p_sign_posn = 3	1.25+ $	1.25 +$	1.25+$
	p_sign_posn = 4	1.25$ +	1.25 $+	1.25$+

The following is an example of the interpretation of the `mon_grouping` keyword. Assuming that the value to be formatted is `123456789` and the `mon_thousands_sep` is `'`, then the following table shows the result. The third column shows the equivalent C Standard {7} string that would be used to accommodate this grouping. It is the responsibility of the utility to perform mappings of the formats in this clause to those used by language bindings such as the C Standard {7}.

mon_grouping	Formatted Value	C Standard {7} String
3;-1	123456'789	"\3\177"
3	123'456'789	"\3"
3;2;-1	1234'56'789	"\3\2\177"
3;2	12'34'56'789	"\3\2"
-1	123456789	"177"

In these examples, the octal value of {CHAR_MAX} is 177.

2.5.2.4 LC_NUMERIC

The LC_NUMERIC category shall define the rules and symbols that shall be used to format nonmonetary numeric information. The operands are strings. For some keywords, the strings only can contain integers. Keywords that are not provided, string values set to the empty string (`""`), or integer keywords set to −1, shall be used to indicate that the value is unspecified. The following keywords shall be recognized:

`copy` Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.

`decimal_point` The operand is a string containing the symbol that shall be used as the decimal delimiter in numeric, nonmonetary formatted quantities. This keyword cannot be omitted and cannot be set to the empty string. In contexts where other standards limit the `decimal_point` to a single byte, the result of specifying a multibyte operand is unspecified. 2

`thousands_sep`
The operand is a string containing the symbol that shall 2
be used as a separator for groups of digits to the left of the 2
decimal delimiter in numeric, nonmonetary formatted 2
monetary quantities. In contexts where other standards 2
limit the `thousands_sep` to a single byte, the result of 2
specifying a multibyte operand is unspecified. 2

`grouping`
Define the size of each group of digits in formatted nonmonetary quantities. The operand is a sequence of integers separated by semicolons. Each integer specifies the number of digits in each group, with the initial integer defining the size of the group immediately preceding the decimal delimiter, and the following integers defining the
preceding groups. If the last integer is not −1, then the 2
size of the previous group (if any) shall be repeatedly used 2
for the remainder of the digits. If the last integer is −1, 2
then no further grouping shall be performed. 2

Table 2-9 – LC_NUMERIC Category Definition in the POSIX Locale

```
LC_NUMERIC
# This is the POSIX Locale definition for
# the LC_NUMERIC category.
#
decimal_point       "<period>"
thousands_sep       ""
grouping            0
#
END LC_NUMERIC
```
2

2.5.2.4.1 LC_NUMERIC Rationale. *(This subclause is not a part of P1003.2)*

See the rationale for LC_MONETARY (2.5.2.3.1) for a description of the behavior of 1
grouping. 1

2.5.2.5 LC_TIME

The LC_TIME category shall define the interpretation of the field descriptors supported by the `date` utility (see 4.15).

The following mandatory keywords shall be recognized:

`copy`
Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.

`abday`
Define the abbreviated weekday names, corresponding to the `%a` field descriptor. The operand shall consist of seven semicolon-separated strings. The first string shall be the abbreviated name

Table 2-10 – LC_TIME Category Definition in the POSIX Locale

```
LC_TIME
# This is the POSIX Locale definition for
# the LC_TIME category.
#
# Abbreviated weekday names (%a)
abday   "<S><u><n>";"<M><o><n>";"<T><u><e>";"<W><e><d>";\
        "<T><h><u>";"<F><r><i>";"<S><a><t>"
#
# Full weekday names (%A)
day     "<S><u><n><d><a><y>";"<M><o><n><d><a><y>";\
        "<T><u><e><s><d><a><y>";"<W><e><d><n><e><s><d><a><y>";\
        "<T><h><u><r><s><d><a><y>";"<F><r><i><d><a><y>";\
        "<S><a><t><u><r><d><a><y>"
#
# Abbreviated month names (%b)
abmon   "<J><a><n>";"<F><e><b>";"<M><a><r>";\
        "<A><p><r>";"<M><a><y>";"<J><u><n>";\
        "<J><u><l>";"<A><u><g>";"<S><e><p>";\
        "<O><c><t>";"<N><o><v>";"<D><e><c>"
#
# Full month names (%B)
mon     "<J><a><n><u><a><r><y>";"<F><e><b><r><u><a><r><y>";\
        "<M><a><r><c><h>";"<A><p><r><i><l>";\
        "<M><a><y>";"<J><u><n><e>";\
        "<J><u><l><y>";"<A><u><g><u><s><t>";\
        "<S><e><p><t><e><m><b><e><r>";"<O><c><t><o><b><e><r>";\
        "<N><o><v><e><m><b><e><r>";"<D><e><c><e><m><b><e><r>"
#
# Equivalent of AM/PM (%p)      "AM";"PM"
am_pm   "<A><M>";"<P><M>"
#
# Appropriate date and time representation (%c)
#       "%a %b %e %H:%M:%S %Y"                                              1
d_t_fmt "<percent-sign><a><space><percent-sign><b><space><percent-sign><e>\    1
<space><percent-sign><H><colon><percent-sign><M>\
<colon><percent-sign><S><space><percent-sign><Y>"
#
# Appropriate date representation (%x)   "%m/%d/%y"
d_fmt   "<percent-sign><m><slash><percent-sign><d><slash><percent-sign><y>"
#
# Appropriate time representation (%X)   "%H:%M:%S"
t_fmt   "<percent-sign><H><colon><percent-sign><M><colon><percent-sign><S>"
#
# Appropriate 12-hour time representation (%r) "%I:%M:%S %p"
t_fmt_ampm "<percent-sign><I><colon><percent-sign><M><colon>\
<percent-sign><S> <percent_sign><p>"
#
END LC_TIME
```

of the first day of the week (Sunday), the second the abbreviated name of the second day, and so on.

`day` Define the full weekday names, corresponding to the `%A` field descriptor. The operand shall consist of seven semicolon-separated strings. The first string shall be the full name of the first day of the week (Sunday), the second the full name of the second day, and so on.

`abmon` Define the abbreviated month names, corresponding to the `%b` field descriptor. The operand shall consist of twelve semicolon-separated strings. The first string shall be the abbreviated name of the first month of the year (January), the second the abbreviated name of the second month, and so on.

`mon` Define the full month names, corresponding to the `%B` field descriptor. The operand shall consist of twelve semicolon-separated strings. The first string shall be the full name of the first month of the year (January), the second the full name of the second month, and so on.

`d_t_fmt` Define the appropriate date and time representation, corresponding to the `%c` field descriptor. The operand shall consist of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in Table 2-15.

`d_fmt` Define the appropriate date representation, corresponding to the `%x` field descriptor. The operand shall consist of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in Table 2-15.

`t_fmt` Define the appropriate time representation, corresponding to the `%X` field descriptor. The operand shall consist of a string, and can contain any combination of characters and field descriptors. In addition, the string can contain escape sequences defined in Table 2-15.

`am_pm` Define the appropriate representation of the *ante meridiem* and *post meridiem* strings, corresponding to the `%p` field descriptor. The operand shall consist of two strings, separated by a semicolon. The first string shall represent the *ante meridiem* designation, the last string the *post meridiem* designation.

`t_fmt_ampm`
Define the appropriate time representation in the 12-hour clock format with `am_pm`, corresponding to the `%r` field descriptor. The operand shall consist of a string and can contain any combination of characters and field descriptors. If the string is empty, the 12-hour format is not supported in the locale.

It is implementation defined whether the following optional keywords shall be recognized. If they are not supported, but present in a `localedef` source, they shall be ignored.

`era` Shall be used to define alternate Eras, corresponding to the `%E` field descriptor modifier. The format of the operand is unspecified, but shall support the definition of the `%EC` and `%Ey` field descriptors, and may also define the `era_year` format (`%EY`).

`era_year` Shall be used to define the format of the year in alternate Era format, corresponding to the `%EY` field descriptor.

`era_d_fmt` Shall be used to define the format of the date in alternate Era notation, corresponding to the `%Ex` field descriptor.

`alt_digits` Shall be used to define alternate symbols for digits, corresponding to the `%O` field descriptor modifier. The operand shall consist of semicolon-separated strings. The first string shall be the alternate symbol corresponding with zero, the second string the symbol corresponding with one, and so on. Up to 100 alternate symbol strings can be specified. The `%O` modifier indicates that the string corresponding to the value specified via the field descriptor shall be used instead of the value.

2.5.2.5.1 LC_TIME Rationale. *(This subclause is not a part of P1003.2)*

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 capitalization if the output is going to be used at the beginning of a sentence.

The LC_TIME descriptions of `abday`, `daya`, and `abmon` imply a Gregorian style 1
calendar (7-day weeks, 12-month years, leap years, etc.). Formatting time strings 1
for other types of calendars is outside the scope of this standard. 1

As specified under the `date` command, the field descriptors corresponding to the optional keywords consist of a modifier followed by a traditional field descriptor (for instance `%Ex`). If the optional keywords are not supported by the implementation or are unspecified for the current locale, these field descriptors shall be treated as the traditional field descriptor. For instance, assume the following keywords:

```
alt_digits      "0th";"1st";"2nd";"3rd";"4th";"5th";\      1
                "6th";"7th";"8th";"9th";"10th"             1

d_fmt           "The %Od day of %B in %Y"                  1
```

On 7/4/1776, the `%x` field descriptor would result in ‘‘The 4th day of July in 1776,’’ 1
while 7/14/1789 would come out as ‘‘The 14 day of July in 1789.’’ It can be noted that the above example is for illustrative purposes only; the `%O` modifier is primarily intended to provide for Kanji or Hindi digits in `date` formats.

While it is clear that an alternate year format is required, there is no consensus on the format or the requirements. As a result, while these keywords are reserved, the details are left unspecified. It is expected that National Standards Bodies will provide specifications.

2.5.2.6 LC_MESSAGES

The LC_MESSAGES category shall define the format and values for affirmative and negative responses. The operands shall be strings or extended regular expressions; see 2.8.4. The following keywords shall be recognized:

`copy` Specify the name of an existing locale to be used as the source for the definition of this category. If this keyword is specified, no other keyword shall be specified.

`yesexpr` The operand shall consist of an extended regular expression that describes the acceptable affirmative response to a question expecting an affirmative or negative response.

`noexpr` The operand shall consist of an extended regular expression that describes the acceptable negative response to a question expecting an affirmative or negative response.

Table 2-11 – LC_MESSAGES Category Definition in the POSIX Locale

```
LC_MESSAGES
# This is the POSIX Locale definition for
# the LC_MESSAGES category.
#
yesexpr "<circumflex><left-square-bracket><y><Y><right-square-bracket>"
#
noexpr  "<circumflex><left-square-bracket><n><N><right-square-bracket>"
END LC_MESSAGES
```

2.5.2.6.1 LC_MESSAGES Rationale. *(This subclause is not a part of P1003.2)*

The LC_MESSAGES category is described in 2.6 as affecting the language used by utilities for their output. The mechanism used by the implementation to accomplish this, other than the responses shown here in the locale definition file, is not specified by this version of this standard. The POSIX.1 working group is developing an interface that would allow applications (and, presumably some of the standard utilities) to access messages from various message catalogs, tailored to a user's **LC_MESSAGES** value.

2.5.3 Locale Definition Grammar

The grammar and lexical conventions in this subclause shall together describe the syntax for the locale definition source. The general conventions for this style of grammar are described in 2.1.2. Any discrepancies found between this grammar and other descriptions in this clause shall be resolved in favor of this grammar.

2.5.3.1 Locale Lexical Conventions

The lexical conventions for the locale definition grammar are described in this subclause.

The following tokens shall be processed (in addition to those string constants shown in the grammar):

`LOC_NAME`	A string of characters representing the name of a locale.
`CHAR`	Any single character.
`NUMBER`	A decimal number, represented by one or more decimal digits.
`COLLSYMBOL`	A symbolic name, enclosed between angle brackets. The string shall not duplicate any charmap symbol defined in the current charmap (if any), or a `COLLELEMENT` symbol.
`COLLELEMENT`	A symbolic name, enclosed between angle brackets, which shall not duplicate either any charmap symbol or a `CHARSYMBOL` symbol.
`CHARSYMBOL`	A symbolic name, enclosed between angle brackets, from the current charmap (if any).
`OCTAL_CHAR`	One or more octal representations of the encoding of each byte in a single character. The octal representation consists of an `escape_char` (normally a backslash) followed by two or more octal digits.
`HEX_CHAR`	One or more hexadecimal representations of the encoding of each byte in a single character. The hexadecimal representation consists of an `escape_char` followed by the constant `'x'` and two or more hexadecimal digits.
`DECIMAL_CHAR`	One or more decimal representations of the encoding of each byte in a single character. The decimal representation consists of an `escape_char` and followed by a `'d'` and two or more decimal digits.
`ELLIPSIS`	The string ‘‘`...`’’.
`EXTENDED_REG_EXP`	An extended regular expression as defined in the grammar in 2.8.5.2.

EOL The line termination character `<newline>`.

2.5.3.2 Locale Grammar

This subclause presents the grammar for the locale definition.

```
%token          LOC_NAME
%token          CHAR
%token          NUMBER
%token          COLLSYMBOL COLLELEMENT
%token          CHARSYMBOL OCTAL_CHAR HEX_CHAR DECIMAL_CHAR
%token          ELLIPSIS
%token          EXTENDED_REG_EXP
%token          EOL

%start  locale_definition

%%

locale_definition       : global_statements locale_categories
                        |                   locale_categories
                        ;

global_statements       : global_statements symbol_redefine
                        | symbol_redefine
                        ;

symbol_redefine         : '#escape_char'  CHAR EOL
                        | '#comment_char' CHAR EOL
                        ;

locale_categories       : locale_categories locale_category
                        | locale_category
                        ;

locale_category         : lc_ctype | lc_collate | lc_messages
                        | lc_monetary | lc_numeric | lc_time
                        ;

/*      The following grammar rules are common to all categories */

char_list               : char_list char_symbol
                        | char_symbol
                        ;

char_symbol             : CHAR | CHARSYMBOL
                        | OCTAL_CHAR | HEX_CHAR | DECIMAL_CHAR
                        ;

locale_name             : LOC_NAME
                        | '"' LOC_NAME '"'
                        ;

/*      The following is the LC_CTYPE category grammar */

lc_ctype                : ctype_hdr ctype_keywords         ctype_tlr
                        | ctype_hdr 'copy' locale_name EOL ctype_tlr
                        ;
```

```
ctype_hdr               : 'LC_CTYPE' EOL                                          2
                        ;                                                         2

ctype_keywords          : ctype_keywords ctype_keyword                            2
                        | ctype_keyword                                           2
                        ;                                                         1

ctype_keyword           : charclass_keyword charclass_list EOL                    1
                        | charconv_keyword charconv_list EOL                      1
                        ;                                                         1

charclass_keyword       : 'upper' | 'lower' | 'alpha' | 'digit'                   1
                        | 'alnum' | 'xdigit' | 'space' | 'print'                  1
                        | 'graph' | 'blank' | 'cntrl'                             1
                        ;                                                         1

charclass_list          : charclass_list ';' char_symbol                          2
                        | charclass_list ';' ELLIPSIS ';' char_symbol             1
                        | char_symbol                                             2
                        ;                                                         1

charconv_keyword        : 'toupper'                                               1
                        | 'tolower'                                               1
                        ;                                                         1

charconv_list           : charconv_list ';' charconv_entry                        2
                        | charconv_entry                                          2
                        ;                                                         1

charconv_entry          : '(' char_symbol ',' char_symbol ')'                     1
                        ;                                                         1

ctype_tlr               : 'END' 'LC_CTYPE' EOL                                    2
                        ;                                                         1

/*      The following is the LC_COLLATE category grammar */                       1

lc_collate              : collate_hdr collate_keywords       collate_tlr          2
                        | collate_hdr 'copy' locale_name EOL collate_tlr          2
                        ;                                                         2

collate_hdr             : 'LC_COLLATE' EOL                                        2
                        ;                                                         2

collate_keywords        :                order_statements                         2
                        | opt_statements order_statements                         2
                        ;                                                         1

opt_statements          : opt_statements collating_symbols                        2
                        | opt_statements collating_elements                       2
                        | collating_symbols                                       1
                        | collating_elements                                      1
                        ;                                                         1

collating_symbols       : 'collating-symbol' COLLSYMBOL EOL                       1
                        ;                                                         1

collating_elements      : 'collating-element' COLLELEMENT                         1
                          'from' '"' char_list '"' EOL                            2
                        ;                                                         1
```

```
order_statements        : order_start collation_order order_end                 1
                        ;                                                       1
order_start             : 'order_start' EOL                                     1
                        | 'order_start' order_opts EOL                          1
                        ;                                                       1
order_opts              : order_opts ';' order_opt                              2
                        | order_opt                                             2
                        ;                                                       1
order_opt               : order_opt ',' opt_word                                2
                        | opt_word                                              2
                        ;                                                       1
opt_word                : 'forward' | 'backward' | 'position'                   2
                        ;                                                       1
collation_order         : collation_order collation_entry                       2
                        | collation_entry                                       2
                        ;                                                       1
collation_entry         : COLLSYMBOL EOL                                        1
                        | collation_element weight_list EOL                     1
                        | collation_element             EOL                     2
                        ;                                                       1
collation_element       : char_symbol                                           1
                        | COLLELEMENT                                           1
                        | ELLIPSIS                                              1
                        | 'UNDEFINED'                                           1
                        ;                                                       1
weight_list             : weight_list ';' weight_symbol                         2
                        | weight_list ';'                                       2
                        | weight_symbol                                         2
                        ;                                                       1
weight_symbol           : char_symbol                                           2
                        | COLLSYMBOL                                            1
                        | '"' char_list '"'                                     1
                        | ELLIPSIS                                              1
                        | 'IGNORE'                                              1
                        ;                                                       1
order_end               : 'order_end' EOL                                       1
                        ;                                                       1
collate_tlr             : 'END' 'LC_COLLATE' EOL                                2
                        ;                                                       1
/*      The following is the LC_MESSAGES category grammar */                    1
lc_messages             : messages_hdr messages_keywords      messages_tlr      2
                        | messages_hdr 'copy' locale_name EOL messages_tlr      2
                        ;                                                       2
messages_hdr            : 'LC_MESSAGES' EOL                                     2
                        ;                                                       2
```

```
messages_keywords       : messages_keywords messages_keyword              2
                        | messages_keyword                                2
                        ;                                                 1

messages_keyword        : 'yesexpr' '"' EXTENDED_REG_EXP '"' EOL          2
                        | 'noexpr'  '"' EXTENDED_REG_EXP '"' EOL          2
                        ;                                                 2

messages_tlr            : 'END' 'LC_MESSAGES' EOL                         2
                        ;                                                 1

/*      The following is the LC_MONETARY category grammar */              1

lc_monetary             : monetary_hdr monetary_keywords       monetary_tlr    2
                        | monetary_hdr 'copy' locale_name EOL  monetary_tlr    2
                        ;                                                 2

monetary_hdr            : 'LC_MONETARY' EOL                               2
                        ;                                                 2

monetary_keywords       : monetary_keywords monetary_keyword              2
                        | monetary_keyword                                2
                        ;                                                 1

monetary_keyword        : mon_keyword_string mon_string EOL               1
                        | mon_keyword_char NUMBER EOL                     2
                        | mon_keyword_char '-1'   EOL                     2
                        | mon_keyword_grouping mon_group_list EOL         1
                        ;                                                 1

mon_keyword_string      : 'int_curr_symbol' | 'currency_symbol'           1
                        | 'mon_decimal_point' | 'mon_thousands_sep'       1
                        | 'positive_sign' | 'negative_sign'               1
                        ;                                                 1

mon_string              : '"' char_list '"'                               1
                        | '""'                                            1
                        ;                                                 1

mon_keyword_char        : 'int_frac_digits' | 'frac_digits'               1
                        | 'p_cs_precedes' | 'p_sep_by_space'              1
                        | 'n_cs_precedes' | 'n_sep_by_space'              1
                        | 'p_sign_posn' | 'n_sign_posn'                   1
                        ;                                                 1

mon_keyword_grouping    : 'mon_grouping'                                  1
                        ;                                                 1

mon_group_list          : NUMBER                                          2
                        | mon_group_list ';' NUMBER                       2
                        ;                                                 2

monetary_tlr            : 'END' 'LC_MONETARY' EOL                         2
                        ;                                                 2

/*      The following is the LC_NUMERIC category grammar */               2

lc_numeric              : numeric_hdr numeric_keywords       numeric_tlr  2
                        | numeric_hdr 'copy' locale_name EOL numeric_tlr  2
                        ;                                                 2
```

```
numeric_hdr             : 'LC_NUMERIC' EOL                                    2
                        ;                                                     2

numeric_keywords        : numeric_keywords numeric_keyword                    2
                        | numeric_keyword                                     2
                        ;                                                     1

numeric_keyword         : num_keyword_string num_string EOL                   1
                        | num_keyword_grouping num_group_list EOL             1
                        ;                                                     1

num_keyword_string      : 'decimal_point'                                     1
                        | 'thousands_sep'                                     1
                        ;                                                     1

num_string              : '"' char_list '"'                                   1
                        | '""'                                                1
                        ;                                                     1

num_keyword_grouping    : 'num_grouping'                                      1
                        ;                                                     1

num_group_list          : NUMBER                                              2
                        | num_group_list ';' NUMBER                           2
                        ;                                                     1


numeric_tlr             : 'END' 'LC_NUMERIC' EOL                              2
                        ;                                                     1

/*      The following is the LC_TIME category grammar */                     1

lc_time                 : time_hdr time_keywords          time_tlr            2
                        | time_hdr 'copy' locale_name EOL time_tlr            2
                        ;                                                     1

time_hdr                : 'LC_TIME' EOL                                       2
                        ;                                                     1

time_keywords           : time_keywords time_keyword                          2
                        | time_keyword                                        2
                        ;                                                     1

time_keyword            : time_keyword_name time_list EOL                     2
                        | time_keyword_fmt time_string EOL                    1
                        | time_keyword_opt time_list EOL                      1
                        ;                                                     1

time_keyword_name       : 'abday' | 'day' | 'abmon' | 'mon'                   2
                        ;                                                     1

time_keyword_fmt        : 'd_t_fmt' | 'd_fmt' | 't_fmt' | 'am_pm' | 't_fmt_ampm' 1
                        ;                                                     1

time_keyword_opt        : 'era' | 'era_year' | 'era_d_fmt' | 'alt_digits'    1
                        ;                                                     1

time_list               : time_list ';' time_string                           2
                        | time_string                                         2
                        ;                                                     1

time_string             : '"' char_list '"'                                   1
                        ;                                                     1
```

```
time_tlr                 : 'END' 'LC_TIME' EOL                                   2
                         ;                                                       1
```

2.5.4 Locale Definition Example. *(This subclause is not a part of P1003.2)*

The following is an example of a locale definition file that could be used as input to the `localedef` utility. It assumes that the utility is executed with the `−f` option, naming a *charmap* file with (at least) the following content:

```
CHARMAP
<space>      \x20
<dollar>     \x24
<A>          \101
<a>          \141
<A-acute>    \346
<a-acute>    \365
<A-grave>    \300                                                                1
<a-grave>    \366
<b>          \142
<C>          \103
<c>          \143
<c-cedilla>  \347
<d>          \x64
<H>          \110
<h>          \150
<eszet>      \xb7
<s>          \x73
<z>          \x7a
END CHARMAP
```

It should not be taken as complete or to represent any actual locale, but only to illustrate the syntax.

A further set of examples is offered as part of Annex F.

```
#
LC_CTYPE
lower   <a>;<b>;<c>;<c-cedilla>;<d>;...;<z>
upper   A;B;C;Ç;...;Z
space   \x20;\x09;\x0a;\x0b;\x0c;\x0d                                            1
blank   \040;\011
toupper (<a>,<A>);(b,B);(c,C);(ç,Ç);(d,D);(z,Z)
END LC_CTYPE
#
LC_COLLATE
#
# The following example of collation is based on the proposed                  1
# Canadian standard Z243.4.1-1990, "Canadian Alphanumeric                       1
# Ordering Standard For Character sets of CSA Z234.4 Standard".                 1
# (Other parts of this example locale definition file do not                    1
# purport to relate to Canada, or to any other real culture.)                   1
# The proposed standard defines a 4-weight collation, such that
# in the first pass, characters are compared without regard to
```

```
# case or accents; in second pass, backwards compare without
# regard to case; in the third pass, forward compare without
# regard to diacriticals.  In the 3 first passes, non-alphabetic        2
# characters are ignored; in the fourth pass, only special
# characters are considered, such that "The string that has a
# special character in the lowest position comes first.  If two
# strings have a special character in the same position, the
# collation value of the special character determines ordering.
#
# Only a subset of the character set is used here; mostly to
# illustrate the set-up.
#
                                                                        2
#
collating-symbol <LOW_VALUE>                                            2
collating-symbol <LOWER-CASE>
collating-symbol <SUBSCRIPT-LOWER>
collating-symbol <SUPERSCRIPT-LOWER>
collating-symbol <UPPER-CASE>
collating-symbol <NO-ACCENT>
collating-symbol <PECULIAR>
collating-symbol <LIGATURE>
collating-symbol <ACUTE>
collating-symbol <GRAVE>
# Further collating-symbols follow.
#
# Properly, the standard does not include any multi-character
# collating elements; the one below is added for completeness.
#
collating_element <ch> from <c><h>
collating_element <CH> from <C><H>
collating_element <Ch> from <C><h>
#
order_start forward;backward;forward;forward,position
#
# Collating symbols are specified first in the sequence to allocate
# basic collation values to them, lower that than of any character.

<LOW_VALUE>                                                             2
<LOWER-CASE>
<SUBSCRIPT-LOWER>
<SUPERSCRIPT-LOWER>
<UPPER-CASE>
<NO-ACCENT>
<PECULIAR>
<LIGATURE>
<ACUTE>
<GRAVE>
<RING-ABOVE>
<DIAERESIS>
<TILDE>
# Further collating symbols are given a basic collating value here.
```

```
#
# Here follows special characters.
<space>      IGNORE;IGNORE;IGNORE;<space>
# Other special characters follow here.
#
# Here comes the regular characters.
<a>          <a>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<A>          <a>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<a-acute>    <a>;<ACUTE>;<LOWER-CASE>;IGNORE
<A-acute>    <a>;<ACUTE>;<UPPER-CASE>;IGNORE
<a-grave>    <a>;<GRAVE>;<LOWER-CASE>;IGNORE
<A-grave>    <a>;<GRAVE>;<UPPER-CASE>;IGNORE
<ae>         <a><e>;<LIGATURE><LIGATURE>;<LOWER-CASE><LOWER-CASE>;IGNORE
<AE>         <a><e>;<LIGATURE><LIGATURE>;<UPPER-CASE><UPPER-CASE>;IGNORE
<b>          <b>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<B>          <b>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<c>          <c>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<C>          <c>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
<ch>         <ch>;<NO-ACCENT>;<LOWER-CASE>;IGNORE
<Ch>         <ch>;<NO-ACCENT>;<PECULIAR>;IGNORE
<CH>         <ch>;<NO-ACCENT>;<UPPER-CASE>;IGNORE
#
# As an example, the strings "Bach" and "bach" could be encoded (for
# compare purposes) as:
# "Bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\          2
#            <NO_ACCENT>;<LOW_VALUE>;<UPPER>;<LOWER>;<LOWER>;<NULL>     2
# "bach"  <b>;<a>;<ch>;<LOW_VALUE>;<NO_ACCENT>;<NO_ACCENT>;\          2
#            <NO_ACCENT>;<LOW_VALUE>;<LOWER>;<LOWER>;<LOWER>;<NULL>     2
#
# The two strings are equal in pass 1 and 2, but differ in pass 3.
#
# Further characters follow.
#
UNDEFINED    IGNORE;IGNORE;IGNORE;IGNORE
#
order_end
#
END LC_COLLATE
#
LC_MONETARY
int_curr_symbol     "USD "
currency_symbol     "$"
mon_decimal_point   "."
mon_grouping        3;0
positive_sign       ""
negative_sign       "-"
p_cs_precedes       1
n_sign_posn         0
END LC_MONETARY
#
LC_NUMERIC
copy "US_en.ASCII"                                                     1
```

```
END LC_NUMERIC
#
LC_TIME
abday     "Sun";"Mon";"Tue";"Wed";"Thu";"Fri";"Sat"
#
day       "Sunday";"Monday";"Tuesday";"Wednesday";\
          "Thursday";"Friday";"Saturday"
#
abmon     "Jan";"Feb";"Mar";"Apr";"May";"Jun";\
          "Jul";"Aug";"Sep";"Oct";"Nov";"Dec"
#
mon       "January";"February";"March";"April";\
          "May";"June";"July";"August";"September";\
          "October";"November";"December"
#
d_t_fmt   "%a %b %d %T %Z %Y\n"
END LC_TIME
#
LC_MESSAGES
yesexpr   "^([yY][[:alpha:]]*)|(OK)"                              1
#
noexpr    "^[nN][[:alpha:]]*"                                     1
END LC_MESSAGES
```

2.6 Environment Variables

Environment variables defined in this clause affect the operation of multiple utilities and applications. There are other environment variables that are of interest only to specific utilities. Environment variables that apply to a single utility only are defined as part of the utility description. See the Environment Variables subclause of the utility descriptions for information on environment variable usage.

The value of an environment variable is a string of characters, as described in 2.7 in POSIX.1 {8}.

Environment variable names used by the standard utilities shall consist solely of uppercase letters, digits, and the _ (underscore) from the characters defined in 2.4. The namespace of environment variable names containing lowercase letters shall be reserved for applications. Applications can define any environment variables with names from this namespace without modifying the behavior of the standard utilities.

If the following variables are present in the environment during the execution of an application or utility, they are given the meaning described below. They may be put into the environment, or changed, by either the implementation or the user. If they are defined in the utility's environment, the standard utilities assume they have the specified meaning. Conforming applications shall not set these environment variables to have meanings other than as described. See 7.2 and 3.12 for methods of accessing these variables.

HOME A pathname of the user's home directory.

LANG This variable shall determine the locale category for any category not specifically selected via a variable starting with **LC_**. **LANG** and the **LC_** variables can be used by applications to determine the language for messages and instructions, collating sequences, date formats, etc. Additional semantics of this variable, if any, are implementation defined.

LC_ALL This variable shall override the value of the **LANG** variable and the value of any of the other variables starting with **LC_**.

LC_COLLATE This variable shall determine the locale category for character collation information within bracketed regular expressions and for sorting. This environment variable determines the behavior of ranges, equivalence classes, and multicharacter collating elements. Additional semantics of this variable, if any, are implementation defined.

LC_CTYPE This variable shall determine the locale category for character handling functions. This environment variable shall determine the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters), the classification of characters (e.g., alpha, digit, graph), and the behavior of character classes. Additional semantics of this variable, if any, are implementation defined.

LC_MESSAGES This variable shall determine the locale category for processing affirmative and negative responses and the language and cultural conventions in which messages should be written. Additional semantics of this variable, if any, are implementation defined. The language and cultural conventions of diagnostic and informative messages whose format is unspecified by this standard should be affected by the setting of **LC_MESSAGES** .

LC_MONETARY This variable shall determine the locale category for monetary-related numeric formatting information. Additional semantics of this variable, if any, are implementation defined.

LC_NUMERIC This variable shall determine the locale category for numeric formatting (for example, thousands separator and radix character) information. Additional semantics of this variable, if any, are implementation defined.

LC_TIME This variable shall determine the locale category for date and time formatting information. Additional semantics of this variable, if any, are implementation defined.

LOGNAME The user's login name.

PATH The sequence of path prefixes that certain functions and utilities apply in searching for an executable file known only by a filename. The prefixes shall be separated by a colon (**:**). When a nonzero-length prefix is applied to this filename, a slash shall be inserted between the prefix and the filename. A zero-length prefix is an obsolescent feature that indicates the current working directory. It appears as two adjacent colons (**::**), as an initial colon preceding the rest of the list, or as a trailing colon following the rest of the list. A Strictly Conforming POSIX.2 Application shall use an actual pathname (such as `'.'`) to represent the current working directory in **PATH** . The list shall be searched from beginning to end, applying the filename to each prefix, until an executable file with the specified name and appropriate execution permissions is found. If the pathname being sought contains a slash, the search through the path prefixes shall not be performed. If the pathname begins with a slash, the specified path shall be resolved as described in 2.2.2.104. If **PATH** is unset or is set to null, the path search is implementation-defined.

SHELL A pathname of the user's preferred command language interpreter. If this interpreter does not conform to the shell command language in Section 3, utilities may behave differently than described in this standard.

TMPDIR A pathname of a directory made available for programs that need a place to create temporary files.

TERM The terminal type for which output is to be prepared. This information is used by utilities and application programs wishing to exploit special capabilities specific to a terminal. The format and allowable values of this environment variable are unspecified.

TZ Time-zone information. The format is described in POSIX.1 {8}8.1.1.

The environment variables **LANG**, **LC_ALL**, **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, and **LC_TIME** (**LC_***) provide for the support of internationalized applications. The standard utilities shall make use of these environment variables as described in this clause and the individual Environment Variables subclauses for the utilities. If these variables specify locale categories that are not based upon the same underlying code set, the results are unspecified.

For utilities used in internationalized applications, if the **LC_ALL** is not set in the environment or is set to the empty string, and if any of **LC_*** variables is not set in the environment or is set to the empty string, the operational behavior of the utility for the corresponding locale category shall be determined by the setting of the **LANG** environment variable. If the **LANG** environment variable is not set or is set to the empty string, the implementation-defined default locale shall be used.

If **LANG** (or any of the **LC_*** environment variables) contains the value `"C"`, or the value `"POSIX"`, the POSIX Locale shall be selected and the standard utilities shall behave in accordance with the rules in the 2.5.1 for the associated category.

If **LANG** (or any of the **LC_*** environment variables) begins with a slash, it shall be interpreted as the pathname of a file that was created in the output format used by the `localedef` utility; see 4.35.6.3. Referencing such a pathname shall result in that locale being used for the category indicated.

If **LANG** (or any of the **LC_*** environment variables) contains one of a set of implementation-defined values, the standard utilities shall behave in accordance with the rules in a corresponding implementation-defined locale description for the associated category.

If **LANG** (or any of the **LC_*** environment variables) contains a value that the implementation does not recognize, the behavior is unspecified.

Additional criteria for determining a valid locale name are implementation defined.

2.6.1 Environment Variables Rationale. *(This subclause is not a part of P1003.2)*

The standard is worded so that the specified variables *may* be provided to the application. There is no way that the implementation can guarantee that a utility will ever see an environment variable, as a parent process can change the environment for its children. The `env −i` command in this standard and the POSIX.1 {8} *exec* family both offer ways to remove any of these variables from the environment.

The language about locale implies that any utilities written in Standard C and conforming to POSIX.2 must issue the following call:

```
setlocale(LC_ALL, "")
```

If this were omitted, the C Standard {7} specifies that the C Locale would be used.

If any of the environment variables is invalid, it makes sense to default to an implementation-defined, consistent locale environment. It is more confusing for a user to have partial settings occur in case of a mistake. All utilities would then behave in one language/cultural environment. Furthermore, it provides a way of forcing the whole environment to be the implementation-defined default. Disastrous results could occur if a pipeline of utilities partially use the environment variables in different ways. In this case, it would be appropriate for utilities that use **LANG** and related variables to exit with an error if any of the variables are invalid. For example, users typing individual commands at a terminal might want `date` to work if **LC_MONETARY** is invalid as long as **LC_TIME** is valid. Since these are conflicting reasonable alternatives, POSIX.2 leaves the results unspecified if the locale environment variables would not produce a complete locale matching the user's specification.

The locale settings of individual categories cannot be truly independent and still guarantee correct results. For example, when collating two strings, characters must first be extracted from each string (governed by **LC_CTYPE**) before being mapped to collating elements (governed by **LC_COLLATE**) for comparison. That is, if **LC_CTYPE** is causing parsing according to the rules of a large, multibyte code set (potentially returning 20 000 or more distinct character code set values), but **LC_COLLATE** is set to handle only an 8-bit code set with 256 distinct characters, meaningful results are obviously impossible.

The **LC_MESSAGES** variable affects the language of messages generated by the standard utilities. This standard does not provide a means whereby applications can easily be written to perform similar feats. Future versions of POSIX.1 {8} and POSIX.2 are expected to provide both functions and utilities to accomplish multilanguage messaging (using message catalogs), but such facilities were not ready for standardization at the time the initial versions of the standards were developed.

This clause is not a full list of all environment variables, but only those of importance to multiple utilities. Nevertheless, to satisfy some members of the balloting group, here is a list of the other environment variable symbols mentioned in this standard:

Variable	Utility	Variable	Utility	
CDPATH	`cd`	**MAKEFLAGS**	`make`	
COLUMNS	`ls`	**OPTARG**	`getopts`	
DEAD	`mailx`	**OPTIND**	`getopts`	
IFS	`sh`	**PRINTER**	`lp`	1
LPDEST	`lp`	**PS1**	`sh`	
MAIL	`sh`	**PS2**	`sh`	
MAILRC	`mailx`			

The description of **PATH** is similar to that in POSIX.1 {8}, except:

— The behavior of a null prefix is marked obsolescent in favor of using a real pathname. This was done at the behest of some members of the balloting group, who apparently felt it offered a more secure environment, where the current directory would not be selected unintentionally.

— The POSIX.1 {8} *exec* description requires an implementation-defined path search when **PATH** is ''not present.'' POSIX.2 spells out that this means ''unset or set to null.'' Many implementations historically have used a default value of `/bin` and `/usr/bin`. POSIX.2 does not mandate that this default path be identical to that retrieved from `getconf _CS_PATH` because it is likely that a transition to POSIX.2 conformance will see the newly-standardized utilities in another directory that needs to be isolated from some historical applications.

— The POSIX.1 {8} **PATH** description is ambiguous about whether an ''executable file'' means one that has the appropriate permissions for the searching process to execute it. One reading would say that a file with any of the execution bits set on would satisfy the search and that an [EACCES] could be returned at that point. This is not the way historical systems work and POSIX.2 has clarified it to mean that the path search will continue until it finds the name with the execute permissions that would allow the process to execute it. (The case of the [ENOEXEC] error is handled in the text of 3.9.1.1.)

The terminology ''beginning to end'' is used in **PATH** to avoid the noninternationalized ''left to right.'' There is no way to have a colon character embedded within a pathname that is part of the **PATH** variable string. Colon is not a member of the portable filename character set, so this should not be a problem. A portable application can retrieve a default **PATH** value (that will allow access to all the standard utilities) from the system using the command:

```
getconf _CS_PATH
```

See the rationale with `command` for an example of using this.

The **SHELL** variable names the user's preferred shell; it is a guide to applications. There is no direct requirement that that shell conform to this standard—that decision should rest with the user. It is the intention of the developers of this standard that alternative shells be permitted, if the user chooses to develop or acquire one. An operating system that builds its shell into the ''kernel'' in such a manner that alternative shells would be impossible does not conform to the spirit of the standard.

The following environment variables are not currently used by the standard utilities (although they may be by future UPE utilities). Implementations should reserve the names for the following purposes:

EDITOR The name of the user's preferred text file editor. The value of this variable is the name of a utility: either a pathname containing a slash, or a filename to be located using the **PATH** environment variable.

VISUAL The name of the user's preferred ''visual,'' or full-screen, text file editor. The value of this variable is the name of a utility: either a pathname containing a slash, or a filename to be located using the **PATH** environment variable.

The decision to restrict conforming systems to the use of digits, uppercase letters, and underscores for environment variable names allows applications to use lowercase letters in their environment variable names without conflicting with any conforming system.

PROCLANG was added to an earlier draft for internationalized applications, but was removed from the standard because the working group determined that it was not of use.

USER was removed from an earlier draft because it was an unreasonable duplication of **LOGNAME** .

2.7 Required Files

The following directories shall exist on conforming systems and shall be used as described. Strictly Conforming POSIX.2 Applications shall not assume the ability to create files in any of these directories.

`/` The root directory.

`/dev` Contains `/dev/null` and `/dev/tty`, described below.

The following directory shall exist on conforming systems and shall be used as described.

`/tmp` A directory made available for programs that need a place to create temporary files. Applications shall be allowed to create files in this directory, but shall not assume that such files are preserved between invocations of the application.

The following files shall exist on conforming systems and shall be both readable and writable.

`/dev/null` An infinite data source/sink. Data written to `/dev/null` is discarded. Reads from `/dev/null` always return end-of-file (EOF).

`/dev/tty` In each process, a synonym for the controlling terminal associated with the process group of that process, if any. It is useful for programs or shell procedures that wish to be sure of writing messages to or reading data from the terminal no matter how output has been redirected.

2.7.1 Required Files Rationale. *(This subclause is not a part of P1003.2)*

A description of the historical `/usr/tmp` was omitted, removing any concept of differences in emphasis between the `/` and `/usr` versions. The descriptions of `/bin`, `/usr/bin`, `/lib`, and `/usr/lib` were omitted because they are not useful for applications. In an early draft, a distinction was made between *system* and *application* directory usage, but this was not found to be useful.

In Draft 8, `/`, `/dev`, `/local`, `/usr/local`, and `/usr/man` were removed. The directories `/` and `/dev` were restored in Draft 9. It was pointed out by several balloters that the notion of a hierarchical directory structure is key to other information presented in later sections of the standard. (Previously, some had argued that special devices and temporary files could conceivably be handled without a directory structure on some implementations. For example, the system could treat the characters ''`/tmp`'' as a special token that would store files using some non-POSIX file system structure. This notion was rejected by the working group, which requires that all the files in this clause be implemented via POSIX file systems.)

The `/tmp` directory is retained in the standard to accommodate historical applications that assume its availability. Future implementations are encouraged to provide suitable directory names in **TMPDIR** and future applications are encouraged to use the contents of **TMPDIR** for creating temporary files.

The standard files `/dev/null` and `/dev/tty` are required to be both readable and writable to allow applications to have the intended historical access to these files.

2.8 Regular Expression Notation

Editor's Note: The entire rationale for this clause appears at the end of the clause.

Regular Expressions (REs) provide a mechanism to select specific strings from a set of character strings.

Regular expressions are a context-independent syntax that can represent a wide variety of character sets and character set orderings, where these character sets are interpreted according to the current locale. While many regular expressions can be interpreted differently depending on the current locale, many features, such as character class expressions, provide for contextual invariance across locales.

The Basic Regular Expression (BRE) notation and construction rules in 2.8.3 shall apply to most utilities supporting regular expressions. Some utilities, instead, support the Extended Regular Expressions (ERE) described in 2.8.4; any exceptions for both cases are noted in the descriptions of the specific utilities using regular expressions. Both BREs and EREs are supported by the Regular Expression Matching interface in 7.3.

2.8.1 Regular Expression Definitions

For the purposes of this clause, the following definitions apply.

2.8.1.1 entire regular expression: The concatenated set of one or more BREs or EREs that make up the pattern specified for string selection.

2.8.1.2 matched: A sequence of zero or more characters is said to be matched by a BRE or ERE when the characters in the sequence corresponds to a sequence of characters defined by the pattern.

Matching shall be based on the bit pattern used for encoding the character, not on 1
the graphic representation of the character. 1

The search for a matching sequence shall start at the beginning of a string and
stop when the first sequence matching the expression is found, where ‘‘first’’ is
defined to mean ‘‘begins earliest in the string.’’ If the pattern permits a variable
number of matching characters and thus there is more than one such sequence
starting at that point, the longest such sequence shall be matched. For example: 1
the BRE `bb*` matches the second through fourth characters of `abbbc`, and the ERE 1
`(wee|week)(knights|night)` matches all ten characters of `weeknights`. 1

Consistent with the whole match being the longest of the leftmost matches, each 1
subpattern, from left to right, shall match the longest possible string. For this 2
purpose, a null string shall be considered to be longer than no match at all. For 2
example, matching the BRE `\(.*\).*` against `abcdef`, the subexpression (`\1`) is 2
`abcdef`, and matching the BRE `\(a*\)*` against `bc`, the subexpression (`\1`) is the 2
null string. 2

When a multicharacter collating element in a bracket expression (see 2.8.3.2) is 1
involved, the longest sequence shall be measured in characters consumed from 1
the string to be matched; i.e., the collating element counts not as one element, but 1
as the number of characters it matches. 1

2.8.1.3 BRE [ERE] matching a single character: A BRE or ERE that matches either a single character or a single collating element.

Only a BRE or ERE of this type that includes a bracket expression (see 2.8.3.2) can 1
match a collating element. 1

2.8.1.4 BRE [ERE] matching multiple characters: A BRE or ERE that matches a concatenation of single characters or collating elements.

Such a BRE or ERE is made up from a *BRE (ERE) matching a single character* and 1
BRE (ERE) special character s. 1

2.8.2 Regular Expression General Requirements

The requirements in this subclause shall apply to both basic and extended regular expressions.

The use of regular expressions is generally associated with text processing; i.e.,
REs (BREs and EREs) operate on text strings; i.e., zero or more characters followed
by an end-of-string delimiter (typically NUL). Some utilities employing regular
expressions limit the processing to lines; i.e., zero or more characters followed by
a `<newline>`. In the regular expression processing described in this standard,
the `<newline>` character is regarded as an ordinary character. This standard 1
specifies within the individual descriptions of those standard utilities employing 1
regular expressions whether they permit matching of `<newline>`s; if not stated 1
otherwise, the use of literal `<newline>`s or any escape sequence equivalent pro- 1
duces undefined results. 1

The interfaces specified in this standard do not permit the inclusion of a NUL
character in an RE or in the string to be matched. If during the operation of a
standard utility a NUL is included in the text designated to be matched, that NUL 1
may designate the end of the text string for the purposes of matching. 1

When a standard utility or function that uses regular expressions specifies that pattern matching shall be performed without regard to the case (upper- or lower-) of either data or patterns, then when each character in the string is matched against the pattern, not only the character, but also its case counterpart (if any), shall be matched.

The implementation shall support any regular expression that does not exceed 256 bytes in length.

This clause uses the term ‘‘invalid’’ for certain constructs or conditions. Invalid REs shall cause the utility or function using the RE to generate an error condition. When ‘‘invalid’’ is not used, violations of the specified syntax or semantics for REs produce undefined results: this may entail an error, enabling an extended syntax for that RE, or using the construct in error as literal characters to be matched.

2.8.3 Basic Regular Expressions

2.8.3.1 BREs Matching a Single Character or Collating Element

A BRE ordinary character, a special character preceded by a backslash, or a period shall match a single character. A bracket expression shall match a single character or a single collating element.

2.8.3.1.1 BRE Ordinary Characters

An ordinary character is a BRE that matches itself: any character in the supported character set, except for the BRE special characters listed in 2.8.3.1.2.

The interpretation of an ordinary character preceded by a backslash (`\`) is undefined, except for:

(1) The characters `)`, `(`, `{`, and `}`.

(2) The digits 1 through 9 (see 2.8.3.3).

(3) A character inside a bracket expression.

2.8.3.1.2 BRE Special Characters

A *BRE special character* has special properties in certain contexts. Outside of those contexts, or when preceded by a backslash, such a character shall be a BRE that matches the special character itself. The BRE special characters and the contexts in which they have their special meaning are:

`. [ \` The period, left-bracket, and backslash shall be special except when used in a bracket expression (see 2.8.3.2). An expression containing a `[` that is not preceded by a backslash and is not part of a bracket expression produces undefined results.

`*` The asterisk is special except when used

— In a bracket expression,

— As the first character of an entire BRE (after an initial `^`, if any), or

— As the first character of a subexpression (after an initial `^`, if any); see 2.8.3.3.

`^` The circumflex shall be special when used

— As an anchor (see 2.8.3.5) or,

— As the first character of a bracket expression (see 2.8.3.2).

`$` The dollar-sign shall be special when used as an anchor.

2.8.3.1.3 Periods in BREs

A period (`.`), when used outside of a bracket expression, is a BRE that shall match any character in the supported character set except NUL.

2.8.3.2 RE Bracket Expression

A bracket expression (an expression enclosed in square brackets, `[]`) is an RE that matches a single collating element contained in the nonempty set of collating elements represented by the bracket expression.

The following rules and definitions apply to bracket expressions:

(1) A *bracket expression* is either a matching list expression or a nonmatching list expression. It consists of one or more expressions: collating elements, collating symbols, equivalence classes, character classes, or range expressions. Strictly Conforming POSIX.2 Applications shall not use range expressions, but conforming implementations shall support regular expressions containing range expressions. The right-bracket (`]`) shall lose its special meaning and represent itself in a bracket expression if it occurs first in the list [after an initial circumflex (`^`), if any]. Otherwise, it shall terminate the bracket expression, unless it appears in a collating symbol (such as `[.].]`) or is the ending right-bracket for a collating symbol, equivalence class, or character class). The special characters

`.` `*` `[` `\`

(period, asterisk, left-bracket, and backslash, respectively) shall lose their special meaning within a bracket expression.

The character sequences

`[.` `[=` `[:`

(left-bracket followed by a period, equals-sign, or colon) shall be special inside a bracket expression and are used to delimit collating symbols, equivalence class expressions, and character class expressions. These symbols shall be followed by a valid expression and the matching terminating sequence `.]`, `=]`, or `:]`, as described in the following items.

(2) A *matching list* expression specifies a list that shall match any one of the expressions represented in the list. The first character in the list shall not be the circumflex. For example, `[abc]` is an RE that matches any of `a`, `b`, or `c`.

(3) A *nonmatching list* expression begins with a circumflex (`^`), and specifies a list that shall match any character or collating element except for the

expressions represented in the list after the leading circumflex. For example, `[^abc]` is an RE that matches any character or collating element except `a`, `b`, or `c`. The circumflex shall have this special meaning only when it occurs first in the list, immediately following the left-bracket.

(4) A *collating symbol* is a collating element enclosed within bracket-period (`[. .]`) delimiters. Collating elements are defined as described in 2.5.2.2.4. Multicharacter collating elements shall be represented as collating symbols when it is necessary to distinguish them from a list of the individual characters that make up the multicharacter collating element. For example, if the string `ch` is a collating element in the current collation sequence with the associated collating symbol `<ch>`, the expression `[[.ch.]]` shall be treated as an RE matching the character sequence `ch`, while `[ch]` shall be treated as an RE matching `c` or `h`. Collating symbols shall be recognized only inside bracket expressions. This implies that the RE `[[.ch.]]*c` shall match the first through fifth character in the string `chchch`. If the string is not a collating element in the current collating sequence definition, or if the collating element has no characters associated with it (e.g., see the symbol `<HIGH>` in the example collation definition shown in 2.5.2.2.4), the symbol shall be treated as an invalid expression.

(5) An *equivalence class expression* shall represent the set of collating elements belonging to an equivalence class, as described in 2.5.2.2.4. Only primary equivalence classes shall be recognized. The class shall be expressed by enclosing any one of the collating elements in the equivalence class within bracket-equal (`[= =]`) delimiters. For example, if `a`, `à`, and `â` belong to the same equivalence class, then `[[=a=]b]`, `[[=à=]b]`, and `[[=â=]b]` shall each be equivalent to `[aàâb]`. If the collating element does not belong to an equivalence class, the equivalence class expression shall be treated as a *collating symbol*.

(6) A *character class expression* shall represent the set of characters belonging to a character class, as defined in the LC_CTYPE category in the current locale. All character classes specified in the current locale shall be recognized. A character class expression shall be expressed as a character class name enclosed within ‘‘bracket-colon’’ (`[: :]`) delimiters.

Strictly conforming POSIX.2 applications shall only use the following character class expressions, which shall be supported on all conforming implementations:

```
[:alnum:]  [:cntrl:]  [:lower:]  [:space:]
[:alpha:]  [:digit:]  [:print:]  [:upper:]
[:blank:]  [:graph:]  [:punct:]  [:xdigit:]
```

(7) A *range expression* represents the set of collating elements that fall between two elements in the current collation sequence, inclusively. It shall be expressed as the starting point and the ending point separated

by a hyphen (–).

Range expressions shall not be used in Strictly Conforming POSIX.2 Applications because their behavior is dependent on the collating sequence. Range expressions shall be supported by conforming implementations.

In the following, all examples assume the collation sequence specified for the POSIX Locale, unless another collation sequence is specifically defined.

The starting range point and the ending range point shall be a collating
element or collating symbol. An equivalence class expression used as a 2
starting or ending point of a range expression produces unspecified 2
results. The ending range point shall collate equal to or higher than the 2
starting range point; otherwise the expression shall be treated as invalid. The order used is the order in which the collating elements are specified in the current collation definition. One-to-many mappings (see 2.5.2.2) shall not be performed. For example, assuming that the character eszet (β) is placed in the basic collation sequence after `r` and `s`, but before `t`, and that it maps to the sequence `ss` for collation purposes, then the expression `[r–s]` matches only `r` and `s`, but the expression `[s–t]` matches `s`, β, or `t`.

The interpretation of range expressions where the ending range point also is the starting range point of a subsequent range expression is undefined.

The hyphen character shall be treated as itself if it occurs first (after an initial `^`, if any) or last in the list, or as an ending range point in a range expression. As examples, the expressions `[–ac]` and `[ac–]` are equivalent and match any of the characters `a`, `c`, or –; the expressions
`[^–ac]` and `[^ac–]` are equivalent and match any characters except `a`, 1
`c`, or –; the expression `[%––]` matches any of the characters between `%` 1
and – inclusive; the expression `[––@]` matches any of the characters between – and `@`, inclusive; and the expression `[a––@]` is invalid, because the letter `a` follows the symbol `-` in the POSIX Locale. To use a hyphen as the starting range point, it shall either come first in the bracket expression or be specified as a collating symbol. For example:
`[][.–.]-0]`, which matches either a right bracket or any character or 1
collating element that collates between hyphen and 0, inclusive. 1

2.8.3.3 BREs Matching Multiple Characters

The following rules can be used to construct BREs matching multiple characters from BREs matching a single character:

(1) The concatenation of BREs shall match the concatenation of the strings
matched by each component of the BRE. 1

(2) A *subexpression* can be defined within a BRE by enclosing it between the character pairs `\(` and `\)`. Such a subexpression shall match whatever

it would have matched without the `\(` and `\)`, except that anchoring within subexpressions is optional behavior; see 2.8.3.5. Subexpressions can be arbitrarily nested.

(3) The *backreference* expression `\`*n* shall match the same (possibly empty) string of characters as was matched by a subexpression enclosed between `\(` and `\)` preceding the `\`*n*. The character *n* shall be a digit from `1` through `9`, specifying the *n*-th subexpression [the one that begins with the *n*-th `\(` and ends with the corresponding paired `\)`]. The expression is invalid if less than *n* subexpressions precede the `\`*n*. For example, the expression `^\(.*\)\1$` matches a line consisting of two adjacent appearances of the same string, and the expression `\(a\)*\1` fails to match `a`.

(4) When a BRE matching a single character, a subexpression, or a backreference is followed by the special character asterisk (`*`), together with that asterisk it shall match what zero or more consecutive occurrences of the BRE would match. For example, `[ab]*` and `[ab][ab]` are equivalent when matching the string `ab`.

(5) When a BRE matching a single character, a subexpression, or a backreference is followed by an *interval expression* of the format `\{`*m*`\}`, `\{`*m*`,\}`, or `\{`*m*`,`*n*`\}`, together with that interval expression it shall match what repeated consecutive occurrences of the BRE would match. The values of *m* and *n* shall be decimal integers in the range $0 \le m \le n \le$ {RE_DUP_MAX}, where *m* specifies the exact or minimum number of occurrences and *n* specifies the maximum number of occurrences. The expression `\{`*m*`\}` shall match exactly *m* occurrences of the preceding BRE, `\{`*m*`,\}` shall match at least *m* occurrences, and `\{`*m,n*`\}` shall match any number of occurrences between *m* and *n*, inclusive.

For example, in the string `abababccccccd` the BRE `c\{3\}` is matched by characters seven through nine, the BRE `\(ab\)\{4,\}` is not matched at all, and the BRE `c\{1,3\}d` is matched by characters ten through thirteen.

The behavior of multiple adjacent duplication symbols (`*` and intervals) produces undefined results.

2.8.3.4 BRE Precedence

The order of precedence shall be as shown in Table 2-12, from high to low.

2.8.3.5 BRE Expression Anchoring

A BRE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar-sign special characters shall be considered BRE anchors in the following contexts:

(1) A circumflex (`^`) shall be an anchor when used as the first character of an entire BRE. The implementation may treat circumflex as an anchor when

Table 2-12 – BRE Precedence

collation-related bracket symbols	`[= =]` `[: :]` `[. .]`
escaped characters	`\`<*special character*>
bracket expression	`[ ]`
subexpressions/backreferences	`\( \)` `\`*n*
single-character- BRE duplication	`*` `\{`*m*`,`*n*`\}`
concatenation	
anchoring	`^` `$`

used as the first character of a subexpression. The circumflex shall anchor the expression (or optionally subexpression) to the beginning of a string; only sequences starting at the first character of a string shall be matched by the BRE. For example, the BRE `^ab` matches `ab` in the string `abcdef`, but fails to match in the string `cdefab`. The BRE `\(^ab\)` may match the former string. A portable BRE shall escape a leading circumflex in a subexpression to match a literal circumflex.

(2) A dollar-sign (`$`) shall be an anchor when used as the last character of an entire BRE. The implementation may treat a dollar-sign as an anchor when used as the last character of a subexpression. The dollar-sign shall anchor the expression (or optionally subexpression) to the end of the string being matched; the dollar-sign can be said to match the ‘‘end-of-string’’ following the last character.

(3) A BRE anchored by both `^` and `$` shall match only an entire string. For example, the BRE `^abcdef$` matches strings consisting only of `abcdef`.

2.8.4 Extended Regular Expressions

The *extended regular expression* (ERE) notation and construction rules shall apply to utilities defined as using extended regular expressions; any exceptions to the following rules are noted in the descriptions of the specific utilities using EREs.

2.8.4.1 EREs Matching a Single Character or Collating Element

An ERE ordinary character, a special character preceded by a backslash, or a period shall match a single character. A bracket expression shall match a single character or a single collating element. An *ERE matching a single character* enclosed in parentheses shall match the same as the ERE without parentheses would have matched.

2.8.4.1.1 ERE Ordinary Characters

An *ordinary character* is an ERE that matches itself. An ordinary character is any
character in the supported character set, except for the ERE special characters 2
listed in 2.8.4.1.2. The interpretation of an ordinary character preceded by a
backslash (\) is undefined.

2.8.4.1.2 ERE Special Characters

An *ERE special character* has special properties in certain contexts. Outside of 1
those contexts, or when preceded by a backslash, such a character shall be an ERE 1
that matches the special character itself. The extended regular expression special
characters and the contexts in which they shall have their special meaning are:

`. [ \ (` The period, left-bracket, backslash, and left-parenthesis are spe- 1
cial except when used in a bracket expression (see 2.8.3.2).

`* + ? {` The asterisk, plus-sign, question-mark, and left-brace are special
except when used in a bracket expression (see 2.8.3.2). Any of the 2
following uses produce undefined results: 2

— If these characters appear first in an ERE, or immediately fol-
lowing a vertical-line, circumflex, or left-parenthesis.

— If a left-brace is not part of a valid interval expression. 1

`|` The vertical-line is special except when used in a bracket expres-
sion (see 2.8.3.2). A vertical-line appearing first or last in an ERE,
or immediately following a vertical-line or a left-parentheses, pro- 1
duces undefined results. 1

`^` The circumflex shall be special when used 1

— As an anchor (see 2.8.4.6) or, 1

— As the first character of a bracket expression (see 2.8.3.2). 1

`$` The dollar-sign shall be special when used as an anchor. 1

2.8.4.1.3 Periods in EREs

A period (`.`), when used outside of a bracket expression, is an ERE that shall
match any character in the supported character set except NUL. 1

2.8.4.2 ERE Bracket Expression

The rules for ERE Bracket Expressions are the same as for Basic Regular Expres-
sions; see 2.8.3.2.

2.8.4.3 EREs Matching Multiple Characters

The following rules shall be used to construct EREs matching multiple characters from EREs matching a single character:

(1) A *concatenation of EREs* shall match the concatenation of the character sequences matched by each component of the ERE. A concatenation of EREs enclosed in parentheses shall match whatever the concatenation without the parentheses matches. For example, both the ERE `cd` and the ERE `(cd)` are matched by the third and fourth character of the string `abcdefabcdef`.

(2) When an ERE matching a single character, or a concatenation of EREs enclosed in parentheses is followed by the special character plus-sign (+), together with that plus-sign it shall match what one or more consecutive occurrences of the ERE would match. For example, the ERE `b+(bc)` matches the fourth through seventh characters in the string `acabbbcde`. And, `[ab]+` and `[ab][ab]*` are equivalent.

(3) When an ERE matching a single character, or a concatenation of EREs enclosed in parentheses is followed by the special character asterisk (*), together with that asterisk it shall match what zero or more consecutive occurrences of the ERE would match. For example, the ERE `b*c` matches the first character in the string `cabbbcde`, and the ERE `b*cd` matches the third through seventh characters in the string `cabbbcdebbbbbbcdbc`. And, `[ab]*` and `[ab][ab]` are equivalent when matching the string `ab`.

(4) When an ERE matching a single character, or a concatenation of EREs enclosed in parentheses is followed by the special character question-mark (?), together with that question-mark it shall match what zero or one consecutive occurrences of the ERE would match. For example, the ERE `b?c` matches the second character in the string `acabbbcde`.

(5) When an ERE matching a single character, or a concatenation of EREs enclosed in parentheses is followed by an *interval expression* of the format $\{m\}$, $\{m,\}$, or $\{m,n\}$, together with that interval expression it shall match what repeated consecutive occurrences of the ERE would match. The values of m and n shall be decimal integers in the range $0 \le m \le n \le$ {RE_DUP_MAX}, where m specifies the exact or minimum number of occurrences and n specifies the maximum number of occurrences. The expression $\{m\}$ shall match exactly m occurrences of the preceding ERE, $\{m,\}$ shall match at least m occurrences, and $\{m,n\}$ shall match any number of occurrences between m and n, inclusive.

For example, in the string `abababccccccd` the ERE `c{3}` is matched by characters seven through nine, and the ERE `(ab){2,}` is matched by characters one through six.

The behavior of multiple adjacent duplication symbols (+, *, ?, and intervals) produces undefined results.

2.8.4.4 ERE Alternation

Two EREs separated by the special character vertical-line (`|`) shall match a string that is matched by either. For example, the ERE `a((bc)|d)` matches the string `abc` and the string `ad`. Single characters, or expressions matching single characters, separated by the vertical bar and enclosed in parentheses, shall be treated as an ERE matching a single character.

2.8.4.5 ERE Precedence

The order of precedence shall be as shown in Table 2-13, from high to low.

Table 2-13 – ERE Precedence

collation-related bracket symbols	`[= =]` `[: :]` `[. .]`
escaped characters	`\`<*special character*>
bracket expression	`[ ]`
grouping	`( )`
single-character- ERE duplication	`* + ? {`*m*`,`*n*`}`
concatenation	
anchoring	`^` `$`
alternation	`|`

For example, the ERE `abba|cde` matches either the string `abba` or the string `cde` (because concatenation has a higher order of precedence than alternation).

2.8.4.6 ERE Expression Anchoring

An ERE can be limited to matching strings that begin or end a line; this is called *anchoring*. The circumflex and dollar-sign special characters shall be considered ERE anchors in the following contexts:

(1) A circumflex (`^`) shall be an anchor when used anywhere outside a bracket expression. The circumflex shall anchor the (sub)expression to the beginning of a string; only sequences starting at the first character of a string shall be matched by the ERE. For example, the EREs `^ab` and `(^ab)` match `ab` in the string `abcdef`, but fail to match in the string `cdefab`.

(2) A dollar-sign (`$`) shall be an anchor when used anywhere outside a bracket expression. It shall anchor the expression to the end of the string being matched; the dollar-sign can be said to match the ‘‘end-of-string’’ following the last character.

(3) An ERE anchored by both `^` and `$` shall match only an entire string. For example, the EREs `^abcdef$` and `(^abcdef$)` match strings consisting only of `abcdef`.

2.8.5 Regular Expression Grammar

Grammars describing the syntax of both basic and extended regular expressions are presented in this subclause. See the grammar conventions in 2.1.2.

2.8.5.1 BRE/ERE Grammar Lexical Conventions

The lexical conventions for regular expressions shall be as described in this subclause.

Except as noted, the longest possible token or delimiter beginning at a given point shall be recognized.

The following tokens shall be processed (in addition to those string constants shown in the grammar):

`COLL_ELEM` Shall be any single-character collating element, unless it is a `META_CHAR`.

`BACKREF` (Applicable only to basic regular expressions.) Shall be the character string consisting of `'\'` followed by a single-digit numeral, 1 through 9.

`DUP_COUNT` Shall represent a numeric constant. It shall be an integer in the range 0 ≤ `DUP_COUNT` ≤ {RE_DUP_MAX}. This token shall only be recognized when the context of the grammar requires it. At all other times, digits not preceded by `'\'` shall be treated as `ORD_CHAR`.

`META_CHAR` Shall be one of the characters:

- `^` When found first in a bracket expression
- `-` When found anywhere but first (after an initial `^`, if any) or last in a bracket expression, or as the ending range point in a range expression
- `]` When found anywhere but first (after an initial `^`, if any) in a bracket expression.

`L_ANCHOR` (Applicable only to basic regular expressions.) Shall be the character `^` when it appears as the first character of a basic regular expression and when not `QUOTED_CHAR`. The `^` may be recognized as an anchor elsewhere; see 2.8.3.5.

`ORD_CHAR` Shall be a character, other than one of the special characters in `SPEC_CHAR`.

`QUOTED_CHAR` Shall be one of the character sequences:

`\^` `\.` `\*` `\[` `\$` `\\`

`R_ANCHOR` (Applicable only to basic regular expressions). Shall be the character `$` when it appears as the last character of a basic regular expression and when not `QUOTED_CHAR`. The `$` may be recognized as an anchor elsewhere; see 2.8.3.5. 1

`SPEC_CHAR` For basic regular expressions, shall be one of the following special characters:

`.`	Anywhere outside bracket expressions
`\`	Anywhere outside bracket expressions
`[`	Anywhere outside bracket expressions
`^`	When an anchor; see 2.8.3.5 2
`$`	When an anchor; see 2.8.3.5 2
`*`	Anywhere except: first in an entire RE; anywhere in a bracket expression; directly following `\(`; directly following an anchoring `^`.

For extended regular expressions, shall be one of the following special characters found anywhere outside bracket expressions:

`^ . [ $ ( ) | * + ? { \`

(The close-parenthesis shall be considered special in this context only if matched with a preceding open-parenthesis.) 2

2.8.5.2 RE and Bracket Expression Grammar

This subclause presents the grammar for basic regular expressions, including the bracket expression grammar that is common to both BREs and EREs.

```
%token   ORD_CHAR QUOTED_CHAR SPEC_CHAR DUP_COUNT

%token   BACKREF L_ANCHOR R_ANCHOR

%token   Back_open_paren  Back_close_paren
/*           '\('            '\)'                         */

%token   Back_open_brace  Back_close_brace
/*          '\{'              '\}'                      */

/*       The following tokens are for the Bracket Expression
         grammar common to both REs and EREs.             */

%token   COLL_ELEM META_CHAR                                      1

%token   Open_equal Equal_close Open_dot Dot_close Open_colon Colon_close     1
/*          '[='         '=]'      '[.'    '.]'       '[:'        ':]'  */   1

%token   class_name
/*       class_name is a keyword to the LC_CTYPE locale category */
/*       (representing a character class) in the current locale  */
/*       and is only recognized between [: and :]               */

%start   basic_reg_exp
```

```
%%

/*                   -------------------------------------------
                     Basic Regular Expression
                     -------------------------------------------
*/
basic_reg_exp      :          RE_expression
                   | L_ANCHOR
                   |                        R_ANCHOR
                   | L_ANCHOR               R_ANCHOR
                   | L_ANCHOR RE_expression
                   |          RE_expression R_ANCHOR
                   | L_ANCHOR RE_expression R_ANCHOR
                   ;
RE_expression      :              simple_RE
                   | RE_expression simple_RE
                   ;
simple_RE          : nondupl_RE
                   | nondupl_RE RE_dupl_symbol                              1
                   ;
nondupl_RE         : one_character_RE
                   | Back_open_paren RE_expression Back_close_paren
                   | Back_open_paren Back_close_paren
                   | BACKREF
                   ;
/*                                                                          1
                     Note:  This grammar does not permit L_ANCHOR or        1
                     R_ANCHOR inside \( and \) (which implies that ^ and $  1
                     are ordinary characters).  This reflects the semantic  1
                     limits on the application, as noted in 2.8.3.5.        1
                     Implementations are permitted to extend the language to 1
                     interpret ^ and $ as anchors in these locations, and as 1
                     such portable applications shall not use unescaped ^   1
                     and $ in positions inside \( and \) that might be      1
                     interpreted as anchors.                                1
*/                                                                          1
one_character_RE   : ORD_CHAR
                   | QUOTED_CHAR
                   | '.'
                   | bracket_expression
                   ;
RE_dupl_symbol     : '*'
                   | Back_open_brace DUP_COUNT               Back_close_brace
                   | Back_open_brace DUP_COUNT ','           Back_close_brace
                   | Back_open_brace DUP_COUNT ',' DUP_COUNT Back_close_brace
                   ;
/*                   -------------------------------------------
                     Bracket Expression
                     ------------------------------------------
*/
```

```
bracket_expression : '[' matching_list    ']'
                   | '[' nonmatching_list ']'
                   ;

matching_list      : bracket_list
                   ;

nonmatching_list   : '^' bracket_list
                   ;

bracket_list       : follow_list
                   | follow_list '-'                              1
                   ;

follow_list        :             expression_term
                   | follow_list expression_term
                   ;

expression_term    : single_expression
                   | range_expression
                   ;

single_expression  : end_range
                   | character_class                              1
                   ;

range_expression   : start_range end_range
                   | start_range '-'
                   ;

start_range        : end_range '-'
                   ;

end_range          : COLL_ELEM
                   | collating_symbol
                                                                  2
                   ;

collating_symbol   : Open_dot COLL_ELEM Dot_close
                   | Open_dot META_CHAR Dot_close
                   ;

equivalence_class  : Open_equal COLL_ELEM Equal_close
                   ;

character_class    : Open_colon class_name Colon_close            1
                   ;
```

2.8.5.3 ERE Grammar

This subclause presents the grammar for extended regular expressions, excluding the bracket expression grammar.

NOTE: The bracket expression grammar and the associated `%token` lines are identical between BREs and EREs. It has been omitted from the ERE subclause to avoid unnecessary editorial duplication.

```
%token   ORD_CHAR QUOTED_CHAR SPEC_CHAR DUP_COUNT

%start   extended_reg_exp
```

```
%%

/*                  ------------------------------------------
                    Extended Regular Expression
                    ------------------------------------------
*/

extended_reg_exp  :                      anchored_ERE
                  |                      nonanchored_ERE
                  | extended_reg_exp '|' nonanchored_ERE
                  | extended_reg_exp '|' anchored_ERE
                  ;

anchored_ERE      : '^' nonanchored_ERE
                  | '^' nonanchored_ERE '$'
                  |     nonanchored_ERE '$'
                  | '^'
                  |                     '$'
                  | '^'                 '$'
                  ;

nonanchored_ERE   :                 ERE_expression
                  | nonanchored_ERE ERE_expression
                  ;

ERE_expression    : one_character_ERE
                  | '(' extended_reg_exp ')'
                  | ERE_expression ERE_dupl_symbol
                  ;

one_character_ERE : ORD_CHAR
                  | '\' SPEC_CHAR
                  | '.'
                  | bracket_expression
                  ;

ERE_dupl_symbol   : '*'
                  | '+'
                  | '?'
                  | '{' DUP_COUNT               '}'
                  | '{' DUP_COUNT ','           '}'
                  | '{' DUP_COUNT ',' DUP_COUNT '}'
                  ;
```

2.8.6 Regular Expression Notation Rationale. *(This subclause is not a part of P1003.2)*

Editor's Note: Some of the text and headings of this rationale have been rearranged. Moved text has not been diffmarked unless it changed. 1

Rather than repeating the description of regular expressions for each utility supporting REs, the working group preferred a common, comprehensive description of regular expressions in one place. The most common behavior is described here, and exceptions or extensions to this are documented for the respective utilities, if appropriate.

The Basic Regular Expression corresponds to the `ed` or historical `grep` type, and the Extended Regular Expression corresponds to the historical `egrep` type (now `grep -E`).

The text is based on the `ed` description and substantially modified, primarily to aid developers and others in the understanding of the capabilities and limitations of regular expressions. Much of this was influenced by the internationalization requirements.

It should be noted that the definitions in this clause do not cover the `tr` utility (see 4.64); the `tr` syntax does not employ regular expressions.

The specification of regular expressions are particularly important to internationalization, because pattern matching operations are very basic operations in business and other operations. The syntax and rules of regular expressions are intended to be as intuitive as possible, to make them easy to understand and use. The historical rules and behavior do not provide that capability to non-English-language users, and does not provide the necessary support for commonly used characters and language constructs. It was necessary to provide extensions to the historical regular expression syntax and rules, to accommodate other languages. Such modifications were proposed by the UniForum Technical Committee Subcommittee on Internationalization and accepted by the working group. As they are limited to bracket expressions, the rationale for these modifications can be found in 2.8.6.3.2.

2.8.6.1 Regular Expression Definitions Rationale. *(This subclause is not a part of P1003.2)*

The definition of which sequence is matched when several are possible is based on the leftmost-longest rule historically used by deterministic recognizers. This rule is much easier to define and describe, and arguably more useful, than the first-match rule historically used by nondeterministic recognizers. It is thought that dependencies on the choice of rule are rare; carefully-contrived examples are needed to demonstrate the difference.

A formal expression of the leftmost-longest rule is:

> The search is performed as if all possible suffixes of the string were tested for a prefix matching the pattern; the longest suffix containing a matching prefix is chosen, and the longest possible matching prefix of the chosen suffix is identified as the matching sequence.

It is possible to determine what strings correspond to subexpressions by recursively applying the leftmost longest rule to each subexpression, but only with the proviso that the overall match is leftmost longest (see 2.8.1.2). For example, matching `\(ac*\)c*d[ac]*\1` against `acdacaaa` should match `acdacaaa` (with `\1=a`); simply matching the longest match for `\(ac*\)` would yield `\1=ac`, but the overall match would be smaller (`acdac`). In principle, the implementation must examine every possible match and among those that yield the leftmost longest total matches, pick the one that does the longest match for the leftmost subexpression and so on. Note that this means that matching by subexpressions is context dependent: a subexpression within a larger RE may match a different string from the one it would match as an independent RE, and two instances of the same subexpression within the same larger RE may match different lengths even in similar sequences of characters. For example, in the ERE `(a.*b)(a.*b)`, the two identical subexpressions would match four and six characters, respectively, of `accbaccccb`. Thus, it is not possible to hierarchically decompose the matching problem into smaller, independent, matching problems.

Matching is based on the bit pattern used for encoding the character, not on the graphic representation of the character. This means that if a character set contains two or more encodings for a graphic symbol, or if the strings searched contain text encoded in more than one code set, no attempt is made to search for any other representation of the encoded symbol. If that is required, the user can specify equivalence classes containing all variations of the desired graphic symbol.

The definition of ‘‘single character’’ has been expanded to include also collating elements consisting of two or more characters; this expansion is applicable only when a bracket expression is included in the BRE or ERE. An example of such a collating element may be the Dutch ‘‘ij’’,which collates as a ‘‘y.’’ In some encodings, a ligature ‘‘i with j’’ exists *as a character*, and would represent a single-character collating element. In another encoding, no such ligature exists, and the two-character sequence ‘‘ij’’is defined as a multicharacter collating element. Outside brackets, the ‘‘ij’’is treated as a two-character RE and will match the same characters in a string. Historically, a bracket expression only matched a single character. If, however, the bracket expression defines, for example, a range that includes ‘‘ij’’,then this particular bracket expression will also match a sequence of the two characters ‘‘i’’and ‘‘j’’in the string.

2.8.6.2 Regular Expression General Requirements Rationale. *(This subclause is not a part of P1003.2)*

Historically, most regular expression implementations only match lines, not strings. However, that is more an effect of the usage than of an inherent feature of regular expressions itself. Consequently, POSIX.2 does not regard `<newline>`s as special; they are ordinary characters, and both a period and a nonmatching list can match them. Those utilities (like `grep`) that do not allow `<newline>`s to match are responsible for eliminating any `<newline>` from strings before matching against the RE. The *regcomp* () function, however, can provide support for such processing without violating the rules of this clause.

The definition of case-insensitive processing is intended to allow matching of multicharacter collating elements as well as characters. For instance, as each character in the string is matched using both its cases, the RE `[[.Ch.]]`, when matched against `char`, is in reality matched against `ch`, `Ch`, `cH`, and `CH`.

Some implementations of `egrep` have had very limited flexibility in handling complex extended regular expressions. POSIX.2 does not attempt to define the complexity of a BRE or ERE, but does place a lower limit on it—any regular expression must be handled, as long as it can be expressed in 256 bytes or less. (Of course, this does not place an upper limit on the implementation.) There are existing programs using a nondeterministic-recognizer implementation that should have no difficulty with this limit. It is possible that a good approach would be to attempt to use the faster, but more limited, deterministic recognizer for simple expressions and to fall back on the nondeterministic recognizer for those expressions requiring it. Nondeterministic implementations must be careful to observe the 2.8.1.2 rules on which match is chosen; the longest match, not the first match, starting at a given character is used.

The term ‘‘invalid’’ highlights a difference between this clause and some others: 1
POSIX.2 frequently avoids mandating of errors for syntax violations because they 1
can be used by implementors to trigger extensions. However, the authors of the 1
internationalization features of regular expressions desired to mandate errors for 1
certain conditions to identify usage problems or nonportable constructs. These 1
are identified within this rationale as appropriate. The remaining syntax viola- 1
tions have been left implicitly or explicitly undefined. For example, the BRE con- 1
struct `\{1,2,3\}` does not comply with the grammar. A conforming application 1
cannot rely on it producing an error nor matching the literal characters 1
`\{1,2,3\}`. The term ‘‘undefined’’ was used in favor of ‘‘unspecified’’ because 1
many of the situations are considered errors on some implementations and it was 1
felt that consistency throughout the clause was preferable to mixing undefined 1
and unspecified. 1

2.8.6.3 Basic Regular Expressions Rationale. *(This subclause is not a part of P1003.2)*

2.8.6.3.1 BREs Matching a Single Character or Collating Element Rationale. *(This subclause is not a part of P1003.2)*

2.8.6.3.2 RE Bracket Expression Rationale. *(This subclause is not a part of P1003.2)*

If a bracket expression must specify both `-` and `]`, then the `]` must be placed first (after the `^`, if any) and the `-` last within the bracket expression.

Range expressions are, historically, an integral part of regular expressions. However, the requirements of ‘‘natural language behavior’’ and portability does conflict: ranges must be treated according to the current collating sequence, and include such characters that fall within the range based on that collating sequence, regardless of character values. This, however, means that the interpretation will differ depending on collating sequence. If, for instance, one collating sequence defines ‘`ä`’’as a variant of ‘`a`’’,while another defines it as a letter following ‘`z`’’,then the expression `[ä-z]` is valid in the first language and invalid in the second. This kind of ambiguity should be avoided in portable applications, and therefore the working group elected to state that ranges must not be used in strictly conforming applications; however, implementations must support them.

Some historical implementations allow range expressions where the ending range point of one range is also the starting point of the next (for instance `[a-m-o]`). This behavior should not be permitted, but to avoid breaking existing implementations, it is now *undefined* whether it is a valid expression, and how it should be interpreted.

Current practice in awk and lex is to accept escape sequences in bracket expressions as per Table 2-15, while the normal regular expression behavior is to regard such a sequence as consisting of two characters. Allowing the awk/lex behavior in regular expressions would change the normal behavior in an unacceptable way; it is expected that awk and lex will decode escape sequences in regular expressions before passing them to *regcomp*() or comparable routines. Each utility describes the escape sequences it accepts as an exception to the rules in this clause; the list is not the same, for historical reasons.

As noted earlier, the new syntax and rules have been added to accommodate other languages than English. These modifications were proposed by the UniForum Subcommittee on Internationalization and accepted by the working group. The remainder of this clause describes the rationale for these modifications.

Internationalization Requirements

The goal of the internationalization effort was to provide functions and capabilities that matched the capabilities of existing implementations, but that adhered to the user's local customs, rules, and environment. This has also been described as ‘‘removing the ASCII (and English language) bias.’’

In addition, other requirements also influence the standardization efforts, such as *portability*, *extensibility*, and *compatibility*.

In a worldwide environment *portability* carries much weight. Wherever feasible, users should be given the capability to develop code that can execute independently of character set, code set, or language.

Standards must also be *extensible*; to support further development, to allow for local or regional extensions, or to accommodate new concepts (such as multibyte characters).

Compatibility does not only refer to support of existing code, but also to making the new syntax, semantics, and functions compatible with existing environments and implementations.

Internationalization Technical Background

The C Standard {7} (and, by implication, also POSIX) recognizes that the ASCII character set used in historical UNIX system implementations is not adequate outside the Anglo-American language area. It is, however, not enough to remove the ASCII bias; the dependency on Anglo-Saxon conventions and rules must also be broadened to accommodate other cultures, including those that require thousands of characters.

Character sets are defined by their *attributes*; typical attributes are the *encoding*, the *collating sequence*, the *character classification*, and the *case mapping*.

It is also recognized that, even within one language area, several combinations of attributes exist: character set attributes are *mutable* and *combinatory*. So, rather than replacing one straitjacket by another, the proposed standards make character sets *user-definable* and *program-selectable*.

The existence of character set attributes is implicit in regular expressions (REs). This implies that regular expressions must recognize and adapt to the *program-selected* set of attributes.

A program *selects* the appropriate character set (or combination of attributes) using the mechanism described in 2.5. The *definition* of a character set (its attributes) is *external* to an executing program. Many combinations of attributes can exist concurrently. Of particular interest are the following attributes:

(1) *Collating Sequence*. In existing implementations, the *encoded* ASCII ordering matches the *logical* English collating sequence. This correspondence does not exist for all code sets or languages. In addition, many languages employ concepts that have no counterparts in English collation:

(a) In many languages, ordering is based on the concept of *string collation* rather than *character collation* as in English. One of the effects of this is that the ordering is based on *collating elements* rather than on characters. Characters typically map into collating elements:

One-to-one mapping, where a character is also a collating element,

One-to-N mapping, where a single character maps into two or more collating elements (as the German ''ß'' (eszet), which collates as ''ss''),

N-to-one mapping, where two or more characters map into one collating element (as in the Spanish ''ll'', which collates between ''l'' and ''m''; i.e., a word beginning with ''ll'' collates *after* a word beginning with ''lo'').

(b) A common method for adding characters to an alphabet is to use diacritical marks, such as accents or circumflex (` ´ ^). In some languages, this creates a completely new character, collated differently from the Latin ''base.'' In other languages these accented characters are collated as variants of the Latin base letter; i.e., they have the same relative order; they are *equivalent*.

If the strings (words) being compared are equal except for ''accents,'' the strings can be ordered based on a secondary ordering *within* the ''equivalence class.'' For instance, in French, the words ''*tache*'', ''*tâche*'', and ''*tacheter*'' collate in that order.

The C Standard {7} recognizes this; it includes new library functions capable of handling complex collation rules. These functions depend on the setting of the *setlocale*() category LC_COLLATE for a definition of the current collation rules.

(2) *Character Classification*. Character classification and case mapping is another area where each language (or even language area) has its own rules. Although users in different countries can use the same code set,

such as ISO 8859-1 {5}, the definition of what constitutes a letter or an uppercase letter may vary.

The C Standard {7} recognizes this; library functions used to classify characters or perform case mapping depend on the *setlocale* () category LC_CTYPE for a definition of how characters map to character classes.

Internationalization Proposal Areas

Based on the requirements and attribute characteristics defined above, and after reviewing proposals and definitions by X/Open and other organizations, the UniForum Subcommittee on Internationalization decided to concentrate on the following areas: the range expression, character classes, the definition of one-character RE (multicharacter element), and equivalence classes.

Most of these are heavily dependent on the current definition of collation sequence; the Subcommittee felt it natural to couple the capabilities and interpretation of bracket expressions closely to the requirements for extended collation capabilities.

In addition, the Subcommittee felt that the capabilities described in 2.5 formed a suitable basis for runtime control of regular expression behavior.

The Subcommittee realized that the mechanism selected requires changes in the existing syntax. As a rule, the Subcommittee wished to minimize changes and avoid syntactical changes that may cause existing regular expressions to fail.

(1) *Collating Elements and Symbols* . As noted above, many expressions within a bracket expression are closely connected with collation, and the Subcommittee defined many capabilities in terms of collating elements and collating symbols.

A collating element is defined as a sequence of one or more bytes defined in the current collating sequence definition as a unit of collation. In most cases, a collating element is equal to a character, but the collation sequence may exclude some characters, or define two or more characters as a collating element.

A one-character RE is, logically enough, defined as one character or something that translates into one character (the number of bits used to represent the character is not an issue here). The expression within square brackets is a one-character RE; i.e., single characters are matched against the list of single characters defined within the brackets.

In Spanish, the phrase ‘‘a to d’’ means the sequence of collating elements a, á, b, c, ch, and d. Consequently, with a Spanish character set, the range statement [a–d] includes the ch collating element, even though it is expressed with two characters (N-to-1 mapping).

The historical syntax, however, does not allow the user to define either the range from a through ch, or to define ch as a single character rather than as either c or h.

The Subcommittee decided that N-to-1 mappings be recognized (if properly delimited), as *one-character REs* inside, but not outside, square brackets (e.g., a period will never match `ch`).

To be distinguishable from a list of the characters themselves, the multicharacter element must be delimited from the remainder of the characters in the string. The characters `[.` and `.]` are used to delimit a multicharacter collating element from other elements, and can be used to delimit single-character collating elements.

(2) *Equivalence Classes*. As stated previously, many languages extend the Latin alphabet by using diacritical marks. In some cases, the Latin base character (e.g., `a`) and the accented versions of the base (e.g., `à`, `â` in French) constitute a ''subclass'' of characters with some partially equivalent characteristics but different code values. Because these characters are related, they are often processed as a group. The historical syntax, however, does not provide for this in a portable manner.

Although it represents an extension of the historical capabilities, the X/Open group strongly recommended that a properly delimited collating element be recognized as representing an equivalence class, that is as the collating element itself, and all other characters with the same primary order in the collation sequence.

The Subcommittee supported this recommendation, and also selected `[=` and `=]` as delimiters for equivalence classes.

(3) *Range Expressions.* The hyphen historically indicated ''a range of consecutive ASCII characters;'' typically it stands for the word ''to,''as in '`a to z`,'' and implies an ordered *interval*. In ASCII, the *encoded* order matches the *logical* English order; this is not true with other encodings or with other alphabets.

If the ASCII dependency is removed, an alternative could have been to use the encoded sequence of whatever code set is currently used. This, however, would certainly decrease portability, as well as requiring the user to know the ordering of the current code set. It would also most certainly be counter-intuitive; a French user would expect the expression `[a-d]` to match any of the letters `a`, `à` `â`, `b`, `c`, `ç` or `d`. The Subcommittee regards this interpretation of ranges as most compatible with existing capabilities, and one that provides for the desired portability.

As the *logical* ordering need not be inherent in the *encoded* sequence, an external definition was required. Such a definition was already present via the *collating sequence* attribute of the character set. The *setlocale*() function provides for an LC_COLLATE category, which defines the current collating sequence. The Subcommittee selected this as the basis for the interpretation of ranges, as well as of equivalence classes and multicharacter collating symbols.

(4) *Character Classes.* The *range* expression is commonly used to indicate a *character class*; the *ex(au_cmd)* section of the *SVID* states: ''*. . . a pair of*

characters separated by – defines a range (e.g., `a-z` *defines any lowercase letter). . . .*'' In reality, `[a-z]` means ''any lowercase letter between `a` and `z`, inclusive.'' This is *only* equivalent to ''any lowercase letter'' if the *a* is the first and `z` is the last lowercase letter in the collating sequence.

To provide the intended capabilities in a portable way, the Subcommittee introduced a new syntactical element, namely an explicit *character class*. The definition of which characters constitute a specific character class is already present via the LC_CTYPE category of the *setlocale* () function.

The Subcommittee selected the identification of character classes by *name* , bracketed by `[:` and `:]`. A character class cannot be used as an endpoint in a range statement.

Internationalization Syntax

The Subcommittee was careful to propose changes in the regular expression syntax that minimize the impact on existing REs. In evaluating alternatives, the Subcommittee looked at ease of use (terseness, ease to remember, keyboard availability), impact on historical REs (compatibility), implementability, performance and how error-prone the syntax is likely to be (ambiguity).

The Subcommittee made the following evaluation:

(1) Syntax changes must be limited to expressions within square brackets.

(2) Strings or characters with special meaning must be delimited from ordinary strings, to avoid compatibility problems.

(3) Both initial and terminating delimiter should consist of two characters, to minimize compatibility and ambiguity problems.

(4) Outer delimiter character should be bracketing; i.e., naturally indicate initial and terminating side. Examples: `{}` `<>` `()`.

(5) The brackets (`[ ]`) are, due to the special rules for ''brackets within brackets,'' rather unlikely to be used in the intended way (a closing bracket must precede an open bracket in the existing syntax).

(6) To minimize ambiguity, brackets must be paired with another character. Many other symbols are already in use, either within regular expressions, or in the shell. Examples of usable characters are: `=` `.` `:`

(7) Because a multicharacter collating element also can be a member of an equivalence class, different delimiters must be chosen for these two expressions. Also, the character class expression must be distinguishable from, e.g., multicharacter collating symbols; although no historical example is known to the Subcommittee, prudence dictated that character classes be given separate delimiters.

(8) The Subcommittee selected the period as the secondary delimiter for multicharacter collating symbols.

(9) The Subcommittee selected the equals-sign as the secondary delimiter for equivalence classes.

(10) The Subcommittee selected the colon as the secondary delimiter for character classes.

The specific syntax and facilities described in this clause represent a coalescence of proposals and implementations from several vendors. Due to differences in facilities and syntax, it was not possible to take one implementation and codify it. There are now several implementations closely patterned on the existing proposal.

The facilities presented in this clause are described in a manner that does not preclude their use with multibyte character sets. However, no attempt has been made to include facilities specifically intended for such character sets.

The definitions of character classes is tied to the LC_CTYPE definition. The set of character classes defined in the C Standard {7} represents the minimum set of character classes required worldwide, i.e., those required by all implementations. It is the working group's belief that local standards bodies, as well as individual vendors, will provide extensions to the standard in these areas, for instance to provide, for example, Kanji character classes.

In many historical implementations, an *invalid range* is treated as if it consisted of the endpoints only. For example, `[z-a]` is treated as `[za]`. Some implementations treat the above range as `[z]`, and others as `[-az]`. Neither is correct, and the working group decided that this should be treated as an error.

It was proposed that the syntax for bracket expressions be simplified such that the ‘‘extra’’ brackets are not needed if the bracket expression only consists of a character class, an equivalence class, or a collating symbol: ‘‘`[:alpha:]`’’ instead of ‘‘`[[:alpha:]]`’’. To ensure unambiguity, if a bracket expression starts with `:`, `=`, or `.`, then it cannot contain a class expression or a collating symbol (or duplicated characters). In addition, it was also proposed that only valid class or collating symbol expressions be accepted: e.g., `[[:ctrl:]]` is an invalid expression. The working group rejected the proposal. While the syntax `[:alpha:]` may be intuitive to some, the proposal does not allow, e.g., `[:digit:.ch.]`. The alternative, to require additional brackets for the latter case would probably cause more errors than the historical syntax. Requiring erroneous class expressions or collating symbols to make the regular expression invalid may minimize the risks for inadvertent spelling errors. However, at this point it was judged that this would reduce consensus.

Consideration was given to eliminating the `[.ch.]` syntax and providing that collating element should be recognized as such both inside and outside bracket expressions. In addition, consideration was given to defining character classes such that collating elements are included. The working group rejected these proposals. The `[.ch.]` syntax is only required inside bracket expressions due to the fact that a bracket expression historically only matched a single character. If `ch` is a collating element, a range `[a-z]` (if ‘‘`ch`’’ falls within it) matches `ch`. Outside brackets, an expression `ch` is treated as two concatenated characters, matching the string ‘‘`ch`’’. The `[.ch.]` expression is intended to allow the specification of a multicharacter collating element separately from ranges in a bracket expression. Character classes are not intended to include collating elements; there is no requirement that all characters in a multicharacter collating element belong to the same character class (for instance ‘‘Ch’’ is ‘‘alpha’’ but neither ‘‘upper’’ nor ‘‘lower’’). Introducing collating elements in character classes would be nonintuitive.

It was suggested that, because ranges may or may not be meaningful (or even accepted) based on the current collating sequence, they should be eliminated from the syntax (or at least marked obsolescent). It was suggested that, e.g., `[z-a]` should always be or never be an error, regardless of collating sequence. The working group did not wish to eliminate ranges from the syntax. While it is true that ranges may not be universally portable, they are nevertheless a useful and fundamental construct in regular expressions. The regular expression syntax has consciously been extended to provide both increased portability and extended local capabilities. Where supported, ranges must reflect the current collating sequence. The working group instead elected to include range expressions as an implementation requirement, but state that strictly conforming applications (but not, e.g., National-Body-conforming applications) shall not use range expressions. Treating erroneous ranges as invalid points out that these may not be portable across collating sequences; and is better than (silently) making them behave in a way contrary to the intents of the user.

Earlier drafts allowed the use of an equivalence class expression as the starting or ending point of a range expression, such as `[[=e=]-f]`. This now produces unspecified results because it is possible to define the equivalence class as a disjoint set of characters. This example could produce different results on various systems:

— An error.

— The equivalent of `[[=e=]e-f]` (which is the correct portable way to include equivalence class effects in a bracket expression).

— All of the collating elements from the lowest value found in the equivalence class, including any of the elements found between the disjoint values.

Consideration was given to saying that equivalence classes with disjoint elements produce unspecified results at the start or end of a range, but since the application cannot predict which equivalence classes are disjoint, this is no improvement over the more general statement chosen.

It was suggested that, while reference to nonprintable characters is partially supported by the proposed set of character classes, the specificity is not precise enough, and that additional character classes should be supported, e.g., `[:tab:]` or `[:a:]`. The working group rejected this proposal, because this feature would represent a substantial enhancement to the current regular expression syntax, and one that cannot be based on internationalization requirements. It is judged that its inclusion would reduce consensus. A future revision of regular expressions should study the capability to create temporary character classes for use in regular expressions; a ‘‘character class macro facility.’’

2.8.6.3.3 BREs Matching Multiple Characters Rationale. *(This subclause is not a part of P1003.2)*

The limit of nine backreferences to subexpressions in the RE is based on the use of a single digit identifier; increasing this to multiple digits would break historical applications. This does not imply that only nine subexpressions are allowed in REs. The following is a valid BRE with ten subexpressions:

```
\(\(\(ab\)*c\)*d\)\(ef\)*\(gh\)\{2\}\(ij\)*\(kl\)*\(mn\)*\(op\)*\(qr\)*
```

The working group regards the common current behavior, which supports `\`*n*`*`, but not `\`*n*`\{`*min,max*`\}`, or `\(...\)*`, or `\(...\)\{`*min,max*`\}`, as a nonintentional result of a specific implementation, and supports both duplication and interval expressions following subexpressions and backreferences.

2.8.6.3.4 Expression Anchoring Rationale. *(This subclause is not a part of P1003.2)*

Often, the dollar-sign is viewed as matching the ending `<newline>` in text files. This is not strictly true; the `<newline>` is typically eliminated from the strings to be matched and the dollar-sign matches the terminating null character.

The ability of `^`, `$`, and `*` to be nonspecial in certain circumstances may be confusing to some programmers, but this situation was changed only in a minor way from historical practice to avoid breaking many existing scripts. Some consideration was given to making the use of the anchoring characters undefined if not escaped and not at the beginning or end of strings. This would cause a number of historical BREs, such as `2^10`, `$HOME`, and `$1.35`, which relied on the characters being treated literally, to become invalid.

However, one relatively uncommon case was changed to allow an extension used on some implementations. Historically, the BREs `^foo` and `\(^foo\)` did not match the same string, despite the general rule that subexpressions and entire BREs match the same strings. To achieve balloting consensus, POSIX.2 has allowed an extension on some systems to treat these two cases in the same way by declaring that anchoring *may* occur at the beginning or end of a subexpression. Therefore, portable BREs that require a literal circumflex at the beginning or a dollar-sign at the end of a subexpression must escape them. Note that a BRE such as `a\(^bc\)` will either match `a^bc` or nothing on different systems under the POSIX.2 rules.

ERE anchoring has been different from BRE anchoring in all historical systems. An unescaped anchor character has never matched its literal counterpart outside of a bracket expression. Some systems treated `foo$bar` as a valid expression that never matched anything, others treated it as invalid. POSIX.2 mandates the former, valid unmatched behavior.

Some systems have extended the BRE syntax to add alternation. For example, the subexpression `\(foo$\|bar\)` would match either `foo` at the end of the string or `bar` anywhere. The extension is triggered by the use of the undefined `\|` sequence. Because the BRE is undefined for portable scripts, the extending system is free to make other assumptions, such as that the `$` represents the end-of-line anchor in the middle of a subexpression. If it were not for the extension, the

$ would match a literal dollar-sign under the POSIX.2 rules. 1

2.8.6.4 Extended Regular Expressions Rationale. *(This subclause is not a part of P1003.2)*

As with basic regular expressions, the working group decided to make the interpretation of escaped ordinary characters undefined.

The right-parenthesis is not listed as an ERE special character because it is only 1
special in the context of a preceding left-parenthesis. If found without a preced- 1
ing left-parenthesis, the right-parenthesis has no special meaning. 1

Based on objections in several ballots, the *interval expression*, {*m,n*}, has been added to extended regular expressions. Historically, the interval expression has only been supported in some extended regular expression implementations. The working group estimated that the addition of interval expressions to extended regular expressions would not decrease consensus, and would also make basic regular expressions more of a subset of extended regular expressions than in many historical implementations.

It was suggested that, in addition to interval expressions, backreferences (*n*) also should be added to extended regular expressions. This was rejected by the working group as likely to decrease consensus.

In historical implementations, multiple duplication symbols are usually interpreted from left to right and treated as additive. As an example, `a+*b` matches zero or more instances of `a` followed by a `b`. In POSIX.2, multiple duplication symbols are undefined; i.e., they cannot be relied upon for portable applications. One reason for this is to provide some scope for future enhancements; the current syntax is very crowded.

The precedence of operations differs between EREs and those in `lex`; in `lex`, for historical reasons, interval expressions have a lower precedence than concatenation.

2.8.6.5 Regular Expression Grammar Rationale. *(This subclause is not a part of P1003.2)*

None.

2.9 Dependencies on Other Standards

2.9.1 Features Inherited from POSIX.1

This subclause describes some of the features provided by POSIX.1 {8} that are assumed to be globally available by all systems conforming to POSIX.2. This subclause does not attempt to detail all of the POSIX.1 {8} features that are required by all of the utilities and functions defined in this standard; the utility and function descriptions point out additional functionality required to provide the corresponding specific features needed by each.

The following subclauses describe frequently used concepts. Utility and function description statements override these defaults when appropriate.

2.9.1.0.1 Features Inherited from POSIX.1 Rationale. *(This subclause is not a part of P1003.2)*

It has been pointed out that POSIX.2 assumes that a lot of POSIX.1 {8} functionality is present, but never states exactly how much. This is an attempt to clarify the assumptions.

This subclause only covers the ''utilities and functions defined by this standard.'' It does not mandate that the specific POSIX.1 {8} interfaces themselves be available to all application programs. A C language program compiled on a POSIX.2 system is not guaranteed that any of the POSIX.1 {8} functions are accessible. (For example, although UNIX system-based implementations of `ls` will use *stat*() to get file status, a POSIX.2 implementation of `ls` on a 'LONG_NAME_OS-based'' implementation might use the *get_file_attributes*() and the *get_file_time_stamps*() system calls.) POSIX.2 only requires equivalent functionality, not equal means of access. In any event, programs requiring the POSIX.1 {8} system interface should specify that they need POSIX.1 {8} conformance and not hope to achieve it by piggybacking on POSIX.2.

2.9.1.1 Process Attributes

The following process attributes, as described in POSIX.1 {8}, are assumed to be supported for all processes in POSIX.2:

- controlling terminal
- current working directory
- effective group ID
- effective user ID
- file descriptors
- file mode creation mask
- process ID
- process group ID
- real group ID
- real user ID
- root directory
- saved set-group-ID
- saved set-user-ID
- session membership
- supplementary group IDs

A conforming implementation may include additional process attributes.

2.9.1.1.1 Process Attributes Rationale. *(This subclause is not a part of P1003.2)*

The supplementary group IDs requirement is minimal. If {NGROUPS_MAX} is defined to be zero, they are not required. If {NGROUPS_MAX} is greater than zero, the supplementary group IDs are used as described in POSIX.1 {8} in various permission checking operations.

The saved-set-group-ID and saved-set-user-ID requirements are also minimal. If {_POSIX_SAVED_IDS} is defined, they are required; otherwise, they are not.

A controlling terminal is needed to control access to `/dev/tty`.

The file creation semantics of POSIX.2 require the effective group ID, effective user ID, and the file mode creation mask.

Pathname resolution and access permission checks require the current working directory, effective group ID, effective user ID, and root directory.

The `kill` utility requires the effective group ID, effective user ID, process ID, process group ID, real group ID, real user ID, saved set-group-ID, saved set-user-ID, and session membership attributes to perform the various signal addressing and permission checks.

The `id` utility is based on the effective group ID, effective user ID, real group ID, real user ID, and supplementary group IDs.

The following process attributes described in POSIX.1 {8} do not seem to be required by POSIX.2: parent process ID, pending signals, process signal mask, time left until an alarm clock signal, *tms_cstime*, *tms_cutime*, *tms_stime*, and *tms_utime*. There are probably other attributes mentioned in POSIX.1 {8} that are not listed here.

2.9.1.2 Concurrent Execution of Processes

The following functionality of the POSIX.1 {8} *fork*() function shall be available on all POSIX.2 conformant systems:

(1) Independent processes shall be capable of executing independently without either process terminating.

(2) A process shall be able to create a new process with all of the attributes referenced in 2.9.1.1, determined according to the semantics of a call to the POSIX.1 {8} *fork*() function followed by a call in the child process to one of the POSIX.1 {8} *exec* functions.

2.9.1.2.1 Concurrent Execution of Processes Rationale. *(This subclause is not a part of P1003.2)*

The historical functionality of *fork*() is required, which permits the concurrent execution of independent processes. A system with a single thread of process execution is not an appropriate base upon which to build a POSIX.2 system. (This requirement was not explicitly stated in the 1988 POSIX.1, but is included in the current POSIX.1 {8}.)

2.9.1.3 File Access Permissions

The file access control mechanism described by *file access permissions* in 2.2.2.55 applies to all files on a conforming POSIX.2 implementation.

2.9.1.3.1 File Access Permissions Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of file protections and access control is assumed to be handled as in POSIX.1 {8}.

2.9.1.4 File Read, Write, and Creation

When a file is to be read or written, the file shall be opened with an access mode corresponding to the operation to be performed. If file access permissions deny access, the requested operation shall fail.

When a file that does not exist is created, the following POSIX.1 {8} features shall apply unless the utility or function description states otherwise:

(1) The file's user ID is set to the effective user ID of the calling process.

(2) The file's group ID is set to the effective group ID of the calling process or the group ID of the directory in which the file is being created.

(3) The file's permission bits are set to:

S_IROTH | S_IWOTH | S_IRGRP | S_IWGRP | S_IRUSR | S_IWUSR

(see POSIX.1 {8} 5.6.1.2) except that the bits specified by the process's file mode creation mask are cleared.

(4) The *st_atime*, *st_ctime*, and *st_mtime* fields of the file shall be updated as specified in *file times update* in 2.2.2.69.

(5) If the file is a directory, it shall be an empty directory; otherwise the file shall have length zero.

(6) Unless otherwise specified, the file created shall be a regular file.

When an attempt is made to create a file that already exists, the action shall depend on the file type:

(1) For directories and FIFO special files, the attempt shall fail and the utility shall either continue with its operation or exit immediately with a nonzero status, depending on the description of the utility.

(2) For regular files:

(a) The file's user ID, group ID, and permission bits shall not be changed.

(b) The file shall be truncated to zero length.

(c) The *st_ctime* and *st_mtime* fields shall be marked for update.

(3) For other file types, the effect is implementation defined.

When a file is to be appended, the file shall be opened in a manner equivalent to using the O_APPEND flag, without the O_TRUNC flag, in the POSIX.1 {8} *open*() call.

2.9.1.4.1 File Read, Write, and Creation Rationale. *(This subclause is not a part of P1003.2)*

Even though it might be possible for a process to change the mode of a file to match a requested operation and change the mode back to its original state after the operation is completed, utilities are not allowed to do this unless the utility description states otherwise. As an example, the `ed` utility `r` command fails if the file to be read does not exist (even though it could create the file and then read it) or the file permissions do not allow read access [even though it could use the POSIX.1 {8} *chmod*() function to make the file readable before attempting to open the file].

2.9.1.5 File Removal

When a directory that is the root directory or current working directory of any process is removed, the effect is implementation defined. If file access permissions deny access, the requested operation shall fail. Otherwise, when a file is removed:

(1) Its directory entry shall be removed from the file system.

(2) The link count of the file shall be decremented.

(3) If the file is an empty directory (see 2.2.2.43):

(a) If no process has the directory open, the space occupied by the directory shall be freed and the directory shall no longer be accessible.

(b) If one or more processes have the directory open, the directory contents shall be preserved until all references to the file have been closed.

(4) If the file is a directory that is not empty, the *st_ctime* field shall be marked for update.

(5) If the file is not a directory:

(a) If the link count becomes zero:

[1] If no process has the file open, the space occupied by the file shall be freed and the file shall no longer be accessible.

[2] If one or more processes have the file open, the file contents shall be preserved until all references to the file have been closed.

(b) If the link count is not reduced to zero, the *st_ctime* field shall be marked for update.

(6) The *st_ctime* and *st_mtime* fields of the containing directory shall be marked for update.

2.9.1.5.1 File Removal Rationale. *(This subclause is not a part of P1003.2)*

This is intended to be a summary of the POSIX.1 {8} *unlink*() and *rmdir*() requirements needed by POSIX.2.

2.9.1.6 File Time Values

All files have the three time values described by *file times update* in 2.2.2.69.

2.9.1.6.1 File Time Values Rationale. *(This subclause is not a part of P1003.2)*

All three time stamps specified by POSIX.1 {8} are needed for utilities like `find`, `ls`, `make`, `test`, and `touch` to work as expected.

2.9.1.7 File Contents

When a reference is made to the contents of a file, *pathname*, this means the equivalent of all of the data placed in the space pointed to by *buf* when performing the *read*() function calls in the following POSIX.1 {8} operations:

```
while (read (fildes, buf, nbytes) > 0)
        ;
```

If the file is indicated by a pathname *pathname*, the file descriptor shall be determined by the equivalent of the following POSIX.1 operation:

```
fildes = open (pathname, O_RDONLY);
```

The value of *nbytes* in the above sequence is unspecified; if the file is of a type where the data returned by *read*() would vary with different values, the value shall be one that results in the most data being returned.

If the *read*() function calls would return an error, it is unspecified whether the contents of the file are considered to include any data from offsets in the file beyond where the error would be returned.

2.9.1.7.1 File Contents Rationale. *(This subclause is not a part of P1003.2)*

This description is intended to convey the traditional behavior for all types of files. This matches the intuitive meaning for regular files, but the meaning is not always intuitive for other types of files. In particular, for FIFO s, pipes, and terminals it must be clear that the contents are not necessarily static at the time a file is opened, but they include the data returned by a sequence of reads until end-of-file is indicated. This is why the *open* () call is specified, with the O_NONBLOCK flag not set.

Some files, especially character special files, are sensitive to the size of a *read* () request. The contents of the file are those resulting from proper choice of this size.

2.9.1.8 Pathname Resolution

The pathname resolution algorithm described by *pathname resolution* in 2.2.2.104 shall be used by conforming POSIX. 2 implementations. See also *file hierarchy* in 2.2.2.58.

2.9.1.8.1 Pathname Resolution Rationale. *(This subclause is not a part of P1003.2)*

The whole concept of hierarchical file systems and pathname resolution is assumed to be handled as in POSIX .1 {8}.

2.9.1.9 Changing the Current Working Directory 2

When the current working directory (see 2.2.2.159) is to be changed, unless the 2
utility or function description states otherwise, the operation shall succeed unless 2
a call to the POSIX .1 {8} *chdir* () function would fail when invoked with the new 2
working directory pathname as its argument. 2

2.9.1.9.1 Changing the Current Working Directory Rationale. *(This subclause 2
is not a part of P1003.2)* 2

This subclause covers the access permissions and pathname structures involved 2
with changing directories, such as with `cd` or (the UPE -extended) `mailx` utilities. 2

2.9.1.10 Establish the Locale

The functionality of the POSIX .1 {8} *setlocale* () function is assumed to be available on all POSIX. 2 conformant systems; i.e., utilities that require the capability of establishing an international operating environment shall be permitted to set the specified category of the international environment.

2.9.1.10.1 Establish the Locale Rationale. *(This subclause is not a part of P1003.2)*

The entire concept of locale categories such as the **LC_*** variables along with any implementation-defined categories is assumed to be handled as in POSIX.1 {8}.

2.9.1.11 Actions Equivalent to POSIX.1 Functions

Some utility descriptions specify that a utility performs actions equivalent to a POSIX.1 {8} function. Such specifications require only that the external effects be equivalent, not that any effect within the utility and visible only to the utility be equivalent.

2.9.1.11.1 Actions Equivalent to POSIX.1 Functions Rationale. *(This subclause is not a part of P1003.2)*

An objection was received to an earlier draft that said this approach of equivalent functions was unreasonable, as the reader (and the person writing a test suite) would be responsible for interpreting which portions of POSIX.1 {8} were included and which were not. For example, would such intermediate effects as the setting of *errno* be required if the related POSIX.1 {8} function called for that? The answer is no: this standard is only concerned with the end results of functions against the file system and the environment, and not any intermediate values or results visible only to the programmer using the POSIX.1 {8} function in a C (or other high-level language) program.

2.9.2 Concepts Derived from the C Standard

Some of the standard utilities perform complex data manipulation using their own procedure and arithmetic languages, as defined in their Extended Description or Operands subclauses. Unless otherwise noted, the arithmetic and semantic concepts (precision, type conversion, control flow, etc.) are equivalent to those defined in the C Standard {7}, as described in the following subclauses. Note that there is no requirement that the standard utilities be implemented in any particular programming language.

2.9.2.0.1 Concepts Derived from the C Standard Rationale. *(This subclause is not a part of P1003.2)*

This subclause was introduced to answer complaints that there was insufficient detail presented by such utilities as `awk` or `sh` about their procedural control statements and their methods of performing arithmetic functions. Earlier drafts, derived heavily from the original manual pages, contained statements such as ‘‘`for` loops similar to the C Standard {7},’’ which was good enough for a general understanding, but insufficient for a real implementation.

The C Standard {7} was selected as a model because most historical implementations of the standard utilities were written in C. Thus, it is more likely that they will act in a manner desired by POSIX.2 without modification.

Using the C Standard {7} is primarily a notational convenience, so the many ''little languages'' in POSIX.2 would not have to be rigorously described in every aspect. Its selection does not require that the standard utilities be written in Standard C; they could be written in common-usage C, Ada, Pascal, assembler language, or anything else.

The sizes of the various numeric values refer to C-language datatypes that are allowed to be different sizes by the C Standard {7}. Thus, like a C-language application, a shell application cannot rely on their exact size. However, it can rely on their minimum sizes expressed in the C Standard {7}, such as {LONG_MAX} for a *long* type.

2.9.2.1 Arithmetic Precision and Operations

Integer variables and constants, including the values of operands and option-arguments, used by the standard utilities shall be implemented as equivalent to the C Standard {7} *signed long* data type; floating point shall be implemented as equivalent to the C Standard {7} *double* type. Conversions between types shall be as described in the C Standard {7}. All variables shall be initialized to zero if they are not otherwise assigned by the application's input.

Arithmetic operators and functions shall be implemented as equivalent to those in the cited C Standard {7} section, as listed in Table 2-14.

The evaluation of arithmetic expressions shall be equivalent to that described in the C Standard {7} section 3.3 Expressions.

2.9.2.2 Mathematic Functions

Any mathematic functions with the same names as those in the C Standard {7}'s sections:

4.5 *Mathematics* <math.h>

4.10.2 *Pseudo-random sequence generation functions*

shall be implemented to return the results equivalent to those returned from a call to the corresponding C function described in the C Standard {7}.

Table 2-14 – C Standard Operators and Functions

<table>
<tr><th>Operation</th><th>C Standard {7} Equivalent Reference</th></tr>
<tr><td>( )</td><td>3.3.1 Primary Expressions</td></tr>
<tr><td>postfix ++
postfix −−</td><td>3.3.2 Postfix Operators</td></tr>
<tr><td>unary +
unary −
prefix ++
prefix −−
~
!
<code>sizeof()</code></td><td>3.3.3 Unary Operators</td></tr>
<tr><td>*
/
%</td><td>3.3.5 Multiplicative Operators</td></tr>
<tr><td>+
−</td><td>3.3.6 Additive Operators</td></tr>
<tr><td><<
>></td><td>3.3.7 Bitwise Shift Operators</td></tr>
<tr><td><, <=
>, >=</td><td>3.3.8 Relational Operators</td></tr>
<tr><td>==
!=</td><td>3.3.9 Equality Operators</td></tr>
<tr><td>&</td><td>3.3.10 Bitwise AND Operator</td></tr>
<tr><td>ˆ</td><td>3.3.11 Bitwise Exclusive OR Operator</td></tr>
<tr><td>|</td><td>3.3.12 Bitwise Inclusive OR Operator</td></tr>
<tr><td>&&</td><td>3.3.13 Logical AND Operator</td></tr>
<tr><td>||</td><td>3.3.14 Logical OR Operator</td></tr>
<tr><td>expr?expr:expr</td><td>3.3.15 Conditional Operator</td></tr>
<tr><td>=, *=, /=, %=, +=, −=
<<=, >>=, &=, ˆ=, |=</td><td>3.3.16 Assignment Operators</td></tr>
<tr><td><code>if ( )</code>
<code>if ( ) ... else</code>
<code>switch ( )</code></td><td>3.6.4 Selection Statements</td></tr>
<tr><td><code>while ( )</code>
<code>do ... while ( )</code>
<code>for ( )</code></td><td>3.6.5 Iteration Statements</td></tr>
<tr><td><code>goto</code>
<code>continue</code>
<code>break</code>
<code>return</code></td><td>3.6.6 Jump Statements</td></tr>
</table>

2.10 Utility Conventions

2.10.1 Utility Argument Syntax

This subclause describes the argument syntax of the standard utilities and introduces terminology used throughout the standard for describing the arguments processed by the utilities.

Within the standard, a special notation is used for describing the syntax of a utility's arguments. Unless otherwise noted, all utility descriptions use this notation, which is illustrated by this example (see 3.9.1):

`utility_name [−a][−b][−c` *option_argument* `][−d|−e]`
`[−f`*option_argument*`][`*operand* ...`]`

The notation used for the Synopsis subclauses imposes requirements on the implementors of the standard utilities and provides a simple reference for the reader of the standard.

(1) The utility in the example is named `utility_name`. It is followed by *options*, *option-arguments*, and *operands*. The arguments that consist of hyphens and single letters or digits, such as `−a`, are known as *options* (or, historically, *flags*). Certain options are followed by an *option-argument*, as shown with `[−c` *option_argument* `]`. The arguments following the last options and option-arguments are named *operands*.

(2) Option-arguments are sometimes shown separated from their options by `<blanks>`, sometimes directly adjacent. This reflects the situation that in some cases an option-argument is included within the same argument string as the option; in most cases it is the next argument. The Utility Syntax Guidelines in 2.10.2 require that the option be a separate argument from its option-argument, but there are some exceptions in this standard to ensure continued operation of historical applications:

(a) If the Synopsis of a standard utility shows a `<space>` between an option and option-argument (as with `[−c` *option_argument* `]` in the example), a conforming application shall use separate arguments for that option and its option-argument.

(b) If a `<space>` is not shown (as with `[−f`*option_argument* `]` in the example), a conforming application shall place an option and its option-argument directly adjacent in the same argument string, without intervening `<blank>`s.

(c) Notwithstanding the requirements on conforming applications, a conforming implementation shall permit, but shall not require, an application to specify options and option-arguments as separate arguments whether or not a `<space>` is shown on the synopsis line.

(d) A standard utility may also be implemented to operate correctly when the required separation into multiple arguments is violated by a nonconforming application.

(3) Options are usually listed in alphabetical order unless this would make the utility description more confusing. There are no implied relationships between the options based upon the order in which they appear, unless otherwise stated in the Options subclause, or unless the exception in 2.10.2 guideline 11 applies. If an option that does not have option-arguments is repeated, the results are undefined, unless otherwise stated.

(4) Frequently, names of parameters that require substitution by actual values are shown with embedded underscores. Alternatively, parameters are shown as follows:

<*parameter name*>

The angle brackets are used for the symbolic grouping of a phrase representing a single parameter and shall never be included in data submitted to the utility.

(5) When a utility has only a few permissible options, they are sometimes shown individually, as in the example. Utilities with many flags generally show all of the individual flags (that do not take option-arguments) grouped, as in:

`utility_name [−abcDxyz][−p` *arg* `][`*operand* `]`

Utilities with very complex arguments may be shown as follows:

`utility_name [`*options* `][`*operands* `]`

(6) Unless otherwise specified, whenever an operand or option-argument is or contains a numeric value:

— the number shall be interpreted as a decimal integer.

— numerals in the range 0 to 2 147 483 647 shall be syntactically recognized as numeric values.

— When the utility description states that it accepts negative numbers as operands or option-arguments, numerals in the range −2 147 483 647 to 2 147 483 647 shall be syntactically recognized as numeric values.

This does not mean that all numbers within the allowable range are necessarily semantically correct. A standard utility that accepts an option-argument or operand that is to be interpreted as a number, and for which a range of values smaller than that shown above is permitted by this standard, describes that smaller range along with the description of the option-argument or operand. If an error is generated, the utility's diagnostic message shall indicate that the value is out of the supported range, not that it is syntactically incorrect.

(7) Arguments or option-arguments enclosed in the [and] notation are optional and can be omitted. The [and] symbols shall never be included in data submitted to the utility.

(8) Arguments separated by the **|** vertical bar notation are mutually exclusive. The **|** symbols shall never be included in data submitted to the utility. Alternatively, mutually exclusive options and operands may be listed with multiple Synopsis lines. For example:

> `utility_name −d` **[**`−a`**][**`−c` *option_argument* **][***operand* ... **]**
>
> `utility_name −e` **[**`−b`**][***operand* ... **]**

When multiple synopsis lines are given for a utility, that is an indication that the utility has mutually exclusive arguments. These mutually exclusive arguments alter the functionality of the utility so that only certain other arguments are valid in combination with one of the mutually exclusive arguments. Only one of the mutually exclusive arguments is allowed for invocation of the utility. Unless otherwise stated in an accompanying Options subclause, the relationships between arguments depicted in the Synopsis subclauses are mandatory requirements placed on conforming applications. The use of conflicting mutually exclusive arguments produces undefined results, unless a utility description specifies otherwise. When an option is shown without the **[ ]** brackets, it means that option is required for that version of the Synopsis. However, it is not required to be the first argument, as shown in the example above, unless otherwise stated.

(9) Ellipses (...) are used to denote that one or more occurrences of an option or operand are allowed. When an option or an operand followed by ellipses is enclosed in brackets, zero or more options or operands can be specified. The forms

> `utility_name −f` *option_argument* ... **[***operand* ... **]**
>
> `utility_name` **[**`−g` *option_argument* **]**... **[***operand* ... **]**

indicate that multiple occurrences of the option and its option-argument preceding the ellipses are valid, with semantics as indicated in the Options subclause of the utility. (See also Guideline 11 in 2.10.2.) In the first example, each option-argument requires a preceding `−f` and at least one `−f` *option_argument* must be given.

(10) When the synopsis line is too long to be printed on a single line in this document, the indented lines following the initial line are continuation lines. An actual use of the command would appear on a single logical line.

2.10.1.1 Utility Argument Syntax Rationale. *(This subclause is not a part of P1003.2)*

This is the subclause where the definitions of *option*, *option-argument*, and *operand* come together.

The working group felt that recent trends toward diluting the Synopsis subclauses of historical manual pages to something like:

```
command [options] [operands]
```

were a disservice to the reader. Therefore, considerable effort was placed into rigorous definitions of all the command line arguments and their interrelationships. The relationships depicted in the Synopses are normative parts of this standard; this information is sometimes repeated in textual form, but that is only for clarity within context.

The use of ''undefined'' for conflicting argument usage and for repeated usage of the same option is meant to prevent portable applications from using conflicting arguments or repeated options, unless specifically allowed, as is the case with `ls` (which allows simultaneous, repeated use of the `−C`, `−l`, and `−1` options). Many historical implementations will tolerate this usage, choosing either the first or the last applicable argument, and this tolerance can continue, but portable applications cannot rely upon it. (Other implementations may choose to print usage messages instead.)

The use of ''undefined'' for conflicting argument usage also allows an implementation to make reasonable extensions to utilities where the implementor considers mutually exclusive options according to POSIX.2 to have a sensible meaning and result.

POSIX.2 does not define the result of a utility when an option-argument or operand is not followed by ellipses and the application specifies more than one of that option-argument or operand. This allows an implementation to define valid (although nonstandard) behavior for the utility when more than one such option or operand are specified.

Allowing `<blank>`s after an option (i.e., placing an option and its option-argument into separate argument strings) when the standard does not require it encourages portability of users, while still preserving backward compatibility of scripts. Inserting `<blank>`s between the option and the option-argument is preferred; however, historical usage has not been consistent in this area; therefore, `<blank>`s are required to be handled by all implementations, but implementations are also allowed to handle the historical syntax. Another justification for selecting the multiple-argument method was that the single-argument case is inherently ambiguous when the option-argument can legitimately be a null string.

Wording was also added to explicitly state that digits are permitted as operands and option-arguments. The lower and upper bounds for the values of the numbers used for operands and option-arguments were derived from the C Standard {7} values for {LONG_MIN} and {LONG_MAX}. The requirement on the standard utilities is that numbers in the specified range do not cause a syntax error although the specification of a number need not be semantically correct for a particular operand or option-argument of a utility. For example, the specification of `dd obs=3000000000` would yield undefined behavior for the application and would be a syntax error because the number 3 000 000 000 is outside of the range −2 147 483 647 to +2 147 483 647. On the other hand, `dd obs=2000000000` may cause some error, such as ‘‘blocksize too large,’’ rather than a syntax error.

2.10.2 Utility Syntax Guidelines

The following guidelines are established for the naming of utilities and for the specification of options, option-arguments, and operands. Clause 7.5 describes a function that assists utilities in handling options and operands that conform to these guidelines.

Operands and option-arguments can contain characters not specified in 2.4.

The guidelines are intended to provide guidance to the authors of future utilities. Some of the standard utilities do not conform to all of these guidelines; in those cases, the Options subclauses describe the deviations.

Guideline 1: Utility names should be between two and nine characters, inclusive.

Guideline 2: Utility names should include lowercase letters (the `lower` character classification) from the set described in 2.4 and digits only.

Guideline 3: Each option name should be a single alphanumeric character (the `alnum` character classification) from the set described in 2.4. The `−W` (capital-W) option shall be reserved for vendor extensions.

NOTE: The other alphanumeric characters are subject to standardization in the future, based on historical usage. Implementors should be aware that future POSIX working groups may offer little sympathy to vendors with isolated extensions in conflict with future drafts.

Guideline 4: All options should be preceded by the `'−'` delimiter character.

Guideline 5: Options without option-arguments should be accepted when grouped behind one `'−'` delimiter.

Guideline 6: Each option and option-argument should be a separate argument, except as noted in 2.10.1, item (2).

Guideline 7: Option-arguments should not be optional.

Guideline 8: When multiple option-arguments are specified to follow a single option, they should be presented as a single argument, using commas within that argument or `<blank>`s within that argument to separate them. 2

Guideline 9: All options should precede operands on the command line.

Guideline 10: The argument `"−−"` should be accepted as a delimiter indicating the end of options. Any following arguments should be treated as operands, even if they begin with the `'−'` character. The `"−−"` argument should not be used as an option or as an operand.

Guideline 11: The order of different options relative to one another should not matter, unless the options are documented as mutually exclusive and such an option is documented to override any incompatible options preceding it. If an option that has option-arguments is repeated, the option and option-argument combinations should be interpreted in the order specified on the command line.

Guideline 12: The order of operands may matter and position-related interpretations should be determined on a utility-specific basis.

Guideline 13: For utilities that use operands to represent files to be opened for either reading or writing, the "−" operand should be used only to mean standard input (or standard output when it is clear from context that an output file is being specified).

Any utility claiming conformance to these guidelines shall conform completely to these guidelines, as if these guidelines contained the term ‘‘shall’’ instead of ‘‘should,’’ except that the utility is permitted to accept usage in violation of these guidelines for backward compatibility as long as the required form is also accepted.

Guidelines 1 and 2 are offered as guidance for locales using Latin alphabets. No recommendations are made by this standard concerning utility naming in other locales.

2.10.2.1 Utility Syntax Guidelines Rationale. *(This subclause is not a part of P1003.2)*

This subclause is based on the rules listed in the *SVID* . It was included for two reasons:

(1) The individual utility descriptions in Sections 4, 5, and 6, and Annexes A and C needed a set of common (although not universal) actions on which they could anchor their descriptions of option and operand syntax. Most of the standard utilities actually do use these guidelines, and many of their historical implementations use the *getopt* () function for their parsing. Therefore, it was simpler to cite the rules and merely identify exceptions.

(2) Writers of portable applications need suggested guidelines if the POSIX community is to avoid the chaos of historical UNIX system command syntax.

It is recommended that all *future* utilities and applications use these guidelines to enhance ‘‘user portability.’’ The fact that some historical utilities could not be changed (to avoid breaking existing applications) should not deter this future goal.

The voluntary nature of the guidelines is highlighted by repeated uses of the word *should* throughout. This usage should not be misinterpreted to imply that utilities that claim conformance in their Options subclauses do not always conform.

Guideline 2 recommends the naming of utilities. In 3.9.1, it is further stated that a command used in the shell command language cannot be named with a trailing colon.

Guideline 3 was changed to allow alphanumeric characters (letters and digits) from the character set to allow compatibility with historical usage. Historical practice allows the use of digits wherever practical; and there are no portability issues that would prohibit the use of digits. In fact, from an internationalization viewpoint, digits (being nonlanguage dependent) are preferable over letters (a ‘‘`-2`’’ is intuitively self-explanatory to any user, while in the ‘‘`-f` *filename*’’ the letter `f` is a mnemonic aid only to speakers of Latin based languages where ‘‘filename’’ happens to translate to a word that begins with `f`. Since guideline 3 still retains the word ‘‘single,’’ multidigit options are not allowed. Instances of historical utilities that used them have been marked obsolescent in this standard, with the numbers being changed from option names to option-arguments.

It is difficult to come up with a satisfactory solution to the problem of namespace in option characters. When the POSIX.2 group desired to extend the historical `cc` utility to accept C Standard {7} programs, it found that all of the portable alphabet was already in use by various vendors. Thus, it had to devise a new name, `c89`, rather than something like `cc -X`. There were suggestions that implementors be restricted to providing extensions through various means (such as using a plus-sign as the option delimiter or using option characters outside the alphanumeric set) that would reserve all of the remaining alphanumeric characters for future POSIX standards. These approaches were resisted because they lacked the historical style of UNIX. Furthermore, if a vendor-provided option should become commonly used in the industry, it would be a candidate for standardization. It would be desireable to standardize such a feature using existing practice for the syntax (the semantics can be standardized with any syntax). This would not be possible if the syntax was one reserved for the vendor. However, since the standardization process may lead to minor changes in the semantics, it may prove to be better for a vendor to use a syntax that will not be affected by standardization. As a compromise, the following statements are made by the developers of POSIX.2:

— In future revisions to this standard, and in other POSIX standards, every attempt will be made to develop new utilities and features that conform to the Utility Syntax Guidelines.

— Future extensions and additions to POSIX standards will not use the `-W` (capital W) option. This option is forever reserved to implementors for extensions, in a manner reminiscent of the option's use in historical versions of the `cc` utility. The other alphanumeric characters are subject to standardization in the future, based on historical usage.

Implementors should be cognizant of these intentions and aware that future POSIX working groups will offer little sympathy to vendors with extensions in conflict with future drafts. In the first version of POSIX.2, vendors held a virtual veto power when conflicts arose with their extensions; in the future, POSIX working groups may be less concerned about preserving isolated extensions that conflict with these statements of intent.

Guideline 8 includes the concept of comma-separated lists in a single argument. It is up to the utility to parse such a list itself because *getopt*() just returns the single string. This situation was retained so that certain historical utilities wouldn't violate the guidelines. Applications preparing for international use should be aware of an occasional problem with comma-separated lists: in some locales, the comma is used as the radix character. Thus, if an application is preparing operands for a utility that expects a comma-separated lists, it should avoid generating noninteger values through one of the means that is influenced by setting the **LC_NUMERIC** variable [such as `awk`, `bc`, `printf`, or *printf*()].

Applications calling any utility with a first operand starting with `"-"` should usually specify `"--"`, as indicated by Guideline 10, to mark the end of the options. This is true even if the Synopsis in this standard does not specify any options; implementations may provide options as extensions to this standard. The standard utilities that do not support Guideline 10 indicate that fact in the Options subclause of the utility description.

Guideline 11 was modified to clarify that the order of different options should not matter relative to one another. However, the order of repeated options that also have option-arguments may be significant; therefore, such options are required to be interpreted in the order that they are specified. The `make` utility is an instance of a historical utility that uses repeated options in which the order is significant. Multiple files are specified by giving multiple instances of the `−f` option, for example:

```
make -f common_header -f specific_rules target
```

Guideline 13 does not imply that all of the standard utilities automatically accept the operand `"−"` to mean standard input or output, nor does it specify the actions of the utility upon encountering multiple `"−"` operands. It simply says that, by default, `"−"` operands shall not be used for other purposes in the file reading/writing [but not *stat*()ing, *unlink*()ing, `touch`ing, etc.] utilities. All information concerning actual treatment of the `"−"` operand is found in the individual utility clauses.

An area of concern that was expressed during the balloting process was that as implementations mature implementation-defined utilities and implementation-defined utility options will result. The notion was expressed that there needed to be a standard way, say an environment variable or some such mechanism, to identify implementation-defined utilities separately from standard utilities that may have the same name. It was decided that there already exist several ways of dealing with this situation and that it is outside of the scope of the standard to attempt to standardize in the area of nonstandard items. A method that exists on some historical implementations is the use of the so-called `/local/bin` or `/usr/local/bin` directory to separate local or additional copies or versions of utilities. Another method that is also used is to isolate utilities into completely separate domains. Still another method to ensure that the desired utility is being used is to request the utility by its full pathname. There are, to be sure, many approaches to this situation; the examples given above serve to illustrate that there is more than one.

2.11 Utility Description Defaults

This clause describes all of the subclauses used within the utility clauses in Section 4 and the other sections that describe standard utilities. It describes:

(1) Intended usage of the subclause.

(2) Global defaults that affect all the standard utilities.

2.11.0.1 Utility Description Defaults Rationale. *(This subclause is not a part of P1003.2)*

This clause is arranged with headings in the same order as all the utility descriptions. It is a collection of related and unrelated information concerning:

(1) The default actions of utilities.

(2) The meanings of notations used in the standard that are specific to individual utility subclauses.

Although this material may seem out of place in Section 2, it is important that this information appear before any of the utilities to be described later. Unfortunately, since the utilities are split into multiple major sections (chapters), this information could not be placed into any one of those sections without confusing cross references.

2.11.1 Synopsis

The Synopsis subclause summarizes the syntax of the calling sequence for the utility, including options, option-arguments, and operands. Standards for utility naming are described in 2.10.2; for describing the utility's arguments in 2.10.1.

2.11.2 Description

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

Some utilities in this standard are described in terms of equivalent POSIX.1 {8} functionality. As explained in 1.1, a fully conforming POSIX.1 {8} base is not a prerequisite for this standard. When specific functions are cited, the underlying operating system shall provide 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 standard. See the utility's Exit Status and Consequences of Errors subclauses for all actions associated with errors encountered by the utility.

2.11.3 Options

The Options subclause 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 2.10.2 or describe all deviations. Apparent disagreements between functionality descriptions in the Options and Description (or Extended Description) subclauses are always resolved in favor of the Options subclause.

Each Options subclause that uses the phrase ''The ... utility shall conform to the utility argument syntax guidelines ...'' refers only to the use of the utility as specified by this standard; 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 unrecognized by the implementation, or when a required option-argument is not provided, standard utilities shall issue a diagnostic message to standard error and exit with a nonzero exit status.

Default Behavior: When this subclause 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, shall recognize `"--"` as a first argument to be discarded.

2.11.3.1 Options Rationale. *(This subclause is not a part of P1003.2)*

Although it has not always been possible, the working group has tried to avoid repeating information and therefore reduced the risk that the duplicate explanations are somehow modified to be out of sync.

The requirement for recognizing `--` 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.

2.11.4 Operands

The Operands subclause 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) subclauses are always resolved in favor of the Operands subclause.

If an operand naming a file can be specified as –, which means to use the standard input instead of a named file, this shall be explicitly stated in this subclause. 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 shall process those operands in the order specified in the command line.

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

2.11.4.1 Operands Rationale. *(This subclause is not a part of P1003.2)*

This usage of – is never shown in the Synopsis. Similarly, this usage of –– is never shown.

The requirement for processing operands in command line order is to avoid a ''WeirdNIX'' utility that might choose to sort the input files alphabetically, by size, or by directory order. Although this might be acceptable for some utilities, in general the programmer has a right to know exactly what order will be chosen.

Some of the standard utilities take multiple *file* operands and act as if they were processing the concatenation of those files. For example,

`asa file1 file2` and `cat file1 file2 | asa`

have similar results when questions of file access, errors, and performance are ignored. Other utilities, such as `grep` or `wc`, have completely different results in these two cases. This latter type of utility is always identified in its Description or Operands subclauses, whereas the former is not. Although it might be possible to create a general assertion about the former case, the following points must be addressed:

— Access times for the files might be different in the operand case versus the `cat` case.

— The utility may have error messages that are cognizant of the input file name and this added value should not be suppressed. (As an example, `awk` sets a variable with the file name at each file boundary.)

2.11.5 External Influences

The External Influences subclause describes all input data that is specified by the invoker, data received from the environment, and other files or databases that may be used by the utility. There are four subclauses that contain all the substantive information about external influences; because of this, this level of header is always left blank.

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 requirements of such invoked utilities are not described in the subclause concerning the standard utility that invokes them.

2.11.5.1 Standard Input

The Standard Input subclause describes the standard input of the utility. This subclause is frequently merely a reference to the following subclause, because many utilities treat standard input and input files in the same manner. Unless otherwise stated, all restrictions described in Input Files apply to this subclause as well.

Use of a terminal for standard input may 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 shall not depend on the existence or value of the environment variables defined in this standard, except as provided by this standard.

Default Behavior: When this subclause is listed as ''None,'' it means that the standard input shall not be read when the utility is used as described by this standard.

2.11.5.1.1 Standard Input Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Standard Input Format in previous drafts to better reflect its role in describing the existence and usage of the file, in addition to its format.

2.11.5.2 Input Files

The Input Files subclause 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/initialization files, databases, etc. Commonly-used files are generally described in one place and cross-referenced by other utilities.

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 shall define in the conformance documentation those utilities that are limited by constraints other than file system space, available memory, and other limits specifically cited by this standard, 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 shall also document any limits that they may have in descending the directory tree that are beyond limits cited by this standard.

When a standard utility reads a seekable input file and terminates without an 1
error before it reaches end-of-file, the utility shall ensure that the file offset in the 1
open file description is properly positioned just past the last byte processed by the 1

utility. For files that are not seekable, the state of the file offset in the open file description for that file is unspecified.

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 (e.g., `make`, `read`, `sh`, etc.) 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. 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 2.12 for a description of this notation.

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

2.11.5.2.1 Input Files Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Input File Formats in previous drafts to better reflect its role in describing the existence and usage of the files, in addition to their format.

The description of file offsets answers the question: Are the following three commands equivalent?

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

The answer is that a conforming application cannot assume they are equivalent. The second command is equivalent to the first only when the file is seekable. In the third command, if the file offset in the open file description were not unspecified, `sed` would have to be implemented so that it read from the pipe one byte at a time or it would have to employ some method to seek backwards on the pipe. Such functionality is not defined currently in POSIX.1 {8} and does not exist on all historical systems. Other utilities, such as `head`, `read`, and `sh`, have similar properties, so the restriction is described globally in this clause. A future revision to this standard may require that the standard utilities leave the file offset in a consistent state for pipes as well as regular files.

The description of conformance documentation about file sizes follows many changes of direction by the working group. Originally, there appeared a limit, {ED_FILE_MAX}, that hoped to impose a minimum file size on `ed`, which has been historically limited to relatively small files. This received objections from various members who said that such a limit merely invited sloppy programming; there should be no limits to a ''well-written'' `ed`. Thus, Draft 8 removed the limit and inserted rationale that this meant `ed` would have to process files of virtually unlimited size. (Surprisingly, no objections or comments were received about that sentence.) However, in discussing the matter with representatives of POSIX.3, it turned out that omitting the limit meant that a corresponding test assertion would also be omitted and no test suite could legitimately stress `ed` with large files. It quickly became clear that restrictions applied to other utilities as well and a solution was needed.

It is not possible for this standard to judge which utilities are in the category with arbitrary file size limits; this would impose too much on implementors. Therefore, the burden is placed on implementors to publicly document any limitations and the resulting pressure in the marketplace should keep most implementations adequate for most portable applications. Typically, larger systems would have larger limits than smaller systems, but since price typically follows function, the user can select a machine that handles his/her problems reasonably given such information. The working group considered adding a limit in 2.13.1 for every file-oriented utility, but felt these limits would not actually be used by real applications and would reduce consensus. This is particularly true for utilities, such as possibly `awk` or `yacc`, that might have rather complex limits not directly related to the actual file size.

The definition of *text file* (see 2.2.2.151) is strictly enforced for input to the standard utilities; very few of them list exceptions to the undefined results called for here. (Of course, ‘‘undefined’’ here does not mean that existing implementations necessarily have to change to start indicating error conditions. Conforming applications cannot rely on implementations succeeding or failing when nontext files are used.)

The utilities that allow line continuation are generally those that accept input languages, rather than pure data. It would be unusual for an input line of this type to exceed {LINE_MAX} bytes and unreasonable to require that the implementation allow unlimited accumulation of multiple lines, each of which could reach {LINE_MAX}. Thus, for a portable application the total of all the continued lines in a set cannot exceed {LINE_MAX}.

The format description is intended to be sufficiently rigorous to allow other applications to generate these input files. However, since `<blank>`s 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 realized.

2.11.5.3 Environment Variables

The Environment Variables subclause lists what variables affect the utility's execution.

The entire manner in which environment variables described in this standard affect the behavior of each utility is described in the Environment Variables subclause for that utility, in conjunction with the global effects of the **LANG** and **LC_ALL** environment variables described in 2.6. The existence or value of environment variables described in this standard shall not otherwise affect the specified behavior of the standard utilities. Any effects of the existence or value of environment variables not described by this standard 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 shall be subjected to the path search described for **PATH** in 2.6.

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that the behavior of the utility is not directly affected by environment variables described by this standard when the utility is used as described by this standard.

2.11.5.3.1 Environment Variables Rationale. *(This subclause is not a part of P1003.2)*

The global default text about the **PATH** search is overkill in this version of POSIX.2 (prior to the UPE) because only one of the standard utilities specifies variables in this way—`make`'s `$(CC)`, `$(LEX)`, etc. It is described here mostly in anticipation of its heavier usage in POSIX.2a. The description of **PATH** indicates separately that names including slashes do not apply, so they do not apply here either.

2.11.5.4 Asynchronous Events

The Asynchronous Events subclause lists how the utility reacts to such events as signals and what signals are caught.

Default Behavior: When this subclause is listed as ‘‘Default,’’ or it refers to ‘‘the standard action for all other signals; see 2.11.5.4,’’ it means that the action taken as a result of the signal shall be one of the following:

(1) The action is that inherited from the parent according to the rules of inheritance of signal actions defined in POSIX.1 {8} (see 2.9.1), or

(2) When no action has been taken to change the default, the default action is that specified by POSIX.1 {8}, or

(3) The result of the utility's execution is as if default actions had been taken.

2.11.5.4.1 Asynchronous Events Rationale. *(This subclause is not a part of P1003.2)*

Because there is no language prohibiting it, 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.

2.11.6 External Effects

The External Effects subclause describes the effects of the utility on the operational environment, including the file system. There are three subclauses that contain all the substantive information about external effects; because of this, this level of header is usually left blank.

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 effects of such invoked utilities are not described in the subclause concerning the standard utility that invokes them.

2.11.6.1 Standard Output

The Standard Output subclause describes the standard output of the utility. This subclause is frequently merely a reference to the following subclause, 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 2.12 for a description of this notation.

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

Default Behavior: When this subclause is listed as ‘‘None,’’ it means that the standard output shall not be written when the utility is used as described by this standard.

2.11.6.1.1 Standard Output Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Standard Output Format in previous drafts to better reflect its role in describing the existence and usage of the file, in addition to its format.

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an `awk` or `lex` parser.

2.11.6.2 Standard Error

The Standard Error subclause 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 unspecified by this standard should be affected by the setting of **LC_MESSAGES** .

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

Default Behavior: When this subclause is listed as ‘‘Used only for diagnostic messages,’’ it means that, unless otherwise stated, the diagnostic messages shall be sent to the standard error only when the exit status is nonzero and the utility is used as described by this standard.

When this subclause is listed as ‘‘None,’’ it means that the standard error shall not be used when the utility is used as described in this standard.

2.11.6.2.1 Standard Error Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Standard Error Format in previous drafts to better reflect its role in describing the existence and usage of the file, in addition to its format.

This subclause 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 subclause 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 nonzero exit status; these are specifically noted in their subclauses. 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.

The format descriptions are intended to be sufficiently rigorous to allow post-processing of output by other programs.

2.11.6.3 Output Files

The Output Files subclause 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 (spool, log, audit files, etc.) are not described in this, or any, subclause. 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.

Implementations shall ensure that temporary files, when 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 requirement:

(1) Resources for temporary files other than the namespace (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 files created by the implementation shall be 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.

Record formats are described in a notation similar to that used by the C language function, *printf* (). See 2.12 for a description of this notation.

Default Behavior: When this subclause 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 standard. However, the utility may create or modify system files, such as log files, that are outside of the utility's normal execution environment.

2.11.6.3.1 Output Files Rationale. *(This subclause is not a part of P1003.2)*

This subclause was globally renamed from Output File Formats in previous drafts to better reflect its role in describing the existence and usage of the files, in addition to their format.

The format description is intended to be sufficiently rigorous to allow post-processing of output by other programs, particularly by an `awk` or `lex` parser.

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

2.11.7 Extended Description

The Extended Description subclause 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 Behavior: When this subclause is listed as ''None,'' no further description is necessary.

2.11.8 Exit Status

The Exit Status subclause describes the values the utility shall 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 subclause, conforming implementations shall use those values for the errors described. In some cases, status values are listed more loosely, such as ‘‘>0.’’ A Strictly Conforming POSIX.2 Application shall not rely on any specific value in the range shown and shall be prepared to receive any value in the range.

Unspecified error conditions may be represented by specific values not listed in the standard.

2.11.8.1 Exit Status Rationale. *(This subclause is not a part of P1003.2)*

Note the additional discussion of exit status values in 3.8.2. It describes require- 1
ments for returning exit values > 125. 1

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 Strictly Conforming POSIX.2 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 the standard. In that way, it will have maximum portability, even on implementations with extensions.

The working group is aware that the general nonenumeration of errors makes it difficult to write test suites that test the *incorrect* operation of utilities. There are some historical implementations that have expended effort to provide detailed status messages and a helpful environment to bypass or explain errors, such as prompting, retrying, or ignoring unimportant syntax errors; other implementations have not. Since there is no realistic way to mandate system behavior in cases of undefined application actions or system problems—in a manner acceptable to all cultures and environments—attention has been limited to the correct operation of utilities by the conforming application. Furthermore, the portable application does not need detailed information concerning errors that it caused through incorrect usage or that it cannot correct anyway. The high degree of competition in the emerging POSIX marketplace should ensure that users requiring friendly, resilient environments will be able to purchase such without detailed specification in this standard.

There is no description of defaults for this subclause because all of the standard utilities specify something (or explicitly state ‘‘Unspecified’’) for Exit Status.

2.11.9 Consequences of Errors

The Consequences of Errors subclause describes the effects on the environment, file systems, process state, etc., 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 subclauses; difficulties accessing, creating, reading, or writing files; or, difficulties associated with the privileges of the process.

The following shall 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 shall issue a diagnostic message to standard error and continue processing the next operand in sequence, but the final exit status shall be returned as nonzero.
- If the requested action characterized by an option or option-argument cannot be performed, the utility shall issue a diagnostic message to standard error and the exit status returned shall be nonzero.
- When an unrecoverable error condition is encountered, the utility shall exit with a nonzero exit status.
- A diagnostic message shall be written to standard error whenever an error condition occurs.

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

2.11.9.1 Consequences of Errors Rationale. *(This subclause is not a part of P1003.2)*

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.

In Draft 9, most of the Consequences of Errors subclauses were changed to ‘‘Default.’’ This is due to the more elaborate description of the default case now carried in this subclause and the fact that most of the standard utilities actually use that default.

2.11.10 Rationale

This subclause provides historical perspective and justification of working group actions concerning the utility.

Examples, Usage

This subclause provides examples and usage of the utility. In some cases certain characters are interpreted as special characters to the shell. In the rest of the standard, these characters are shown without escape characters or quoting (see 3.2). In all examples, however, quoting has been used, showing how sample commands (utility names combined with arguments) could be passed correctly to a shell (see `sh` in 4.56) or as a string to the *system* () function.

History of Decisions Made

This subclause provides historical perspective for decisions that were made.

Unresolved Objections

These subclauses were removed from Draft 10. The Unresolved Objections are maintained in a separate list and do not meet ISO editing requirements for an informative annex.

2.11.10.1 Rationale Rationale. *(This subclause is not a part of P1003.2)*

The Rationale subclauses will be moved to Annex E in the final POSIX.2. Some of the subheadings may be collapsed in that document; in these drafts the working group has not always been very rigorous about what is a description of usage versus a history of decisions made, for example. The final rationale will de-emphasize the chronological aspects of working group decisions.

2.12 File Format Notation

The Standard Input, Standard Output, Standard Error, Input Files, and Output Files subclauses of the utility descriptions, when provided, use a syntax to describe the data organization within the files, when that organization is not otherwise obvious. The syntax is similar to that used by the C language *printf* () function, as described in this clause. When used in Standard Input or Input Files subclauses of the utility descriptions, this syntax describes the format that could have been used to write the text to be read, not a format that could be used by the C language *scanf* () function to read the input file.

The description of an individual record is as follows:

"*<format>*", [*<arg1>*, *<arg2>*, . . . , *<argn>*]

The *format* is a character string that contains three types of objects defined below:

characters Characters that are not *escape sequences* or *conversion specifications*, as described below, shall be copied to the output.

escape sequences
Represent nongraphic characters.

conversion specifications
Specifies the output format of each argument. (See below.)

The following characters have the following special meaning in the format string:

" " (An empty character position.) One or more `<blank>` characters.

Δ Exactly one `<space>` character.

The escape-sequences in Table 2-15 depict the associated action on display devices capable of the action.

Each conversion specification shall be introduced by the percent-sign character (%). After the character %, the following shall appear in sequence:

Table 2-15 – Escape Sequences

Escape Sequence	Represents Character	Terminal Action
`\\`	backslash	None.
`\a`	<alert>	Attempts to alert the user through audible or visible notification.
`\b`	<backspace>	Moves the printing position to one column before the current position, unless the current position is the start of a line.
`\f`	<form-feed>	Moves the printing position to the initial printing position of the next logical page.
`\n`	<newline>	Moves the printing position to the start of the next line.
`\r`	<carriage-return>	Moves the printing position to the start of the current line.
`\t`	<tab>	Moves the printing position to the next tab position on the current line. If there are no more tab positions left on the line, the behavior is undefined.
`\v`	<vertical tab>	Moves the printing position to the start of the next vertical tab position. If there are no more vertical tab positions left on the page, the behavior is undefined.

flags Zero or more *flags*, in any order, that modify the meaning of the conversion specification.

field width An optional string of decimal digits to specify a minimum *field width*. For an output field, if the converted value has fewer bytes than the field width, it shall be padded on the left [or right, if the left-adjustment flag (−), described below, has been given] to the field width.

precision Gives the minimum number of digits to appear for the `d`, `o`, `i`, `u`, `x`, or `X` conversions (the field shall be padded with leading zeros), the number of digits to appear after the radix character for the `e` and `f` conversions, the maximum number of significant digits for the `g` conversion; or the maximum number of bytes to be written from a string in `s` conversion. The precision shall take the form of a period (`.`) followed by a decimal digit string; a null digit string shall be treated as zero.

conversion characters
A conversion character (see below) that indicates the type of conversion to be applied.

The *flag* characters and their meanings are:

− The result of the conversion shall be left-justified within the field.

+ The result of a signed conversion always shall begin with a sign (+ or −).

<space> If the first character of a signed conversion is not a sign, a <space> shall be prefixed to the result. This means that if the <space> and + flags both appear, the <space> flag shall be ignored.

\# The value is to be converted to an ‘‘alternate form.’’ For c, d, i, u, and s conversions, the behavior is undefined. For o conversion, it shall increase the precision to force the first digit of the result to be a zero. For x or X conversion, a nonzero result shall have 0x or 0X prefixed to it, respectively. For e, E, f, g and G conversions, the result shall always contain a radix character, even if no digits follow the radix character. For g and G conversions, trailing zeroes shall not be removed from the result as they usually are.

0 For d, i, o, u, x, X, e, E, f, g, and G conversions, leading zeroes (following any indication of sign or base) shall be used to pad to the field width; no space padding shall be performed. If the 0 and − flags both appear, the 0 flag shall be ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag shall be ignored. For other conversions, the behavior is undefined.

Each conversion character shall result in fetching zero or more arguments. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments shall be ignored.

The *conversion characters* and their meanings are:

d,i,o,u,x,X The integer argument shall be written as signed decimal (d or i), unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x and X). The d and i specifiers shall convert to signed decimal in the style [−]*dddd* . The x conversion shall use the numbers and letters 0123456789abcdef and the X conversion shall use the numbers and letters 0123456789ABCDEF. The *precision* component of the argument shall specify the minimum number of digits to appear. If the value being converted can be represented in fewer digits than the specified minimum, it shall be expanded with leading zeroes. The default precision shall be 1. The result of converting a zero value with a precision of 0 shall be no characters. If both the field width and precision are omitted, the implementation may precede and/or follow numeric arguments of types d, i, and u with <blank>s; arguments of type o (octal) may be preceded with leading zeroes.

f The floating point number argument shall be written in decimal notation in the style "[−]*ddd* .*ddd* ", where the number of digits after the radix character (shown here as a decimal point) shall be equal to the *precision* specification. The LC_NUMERIC locale category shall determine the radix character to use in this format. If the *precision* is omitted from the argument, six digits shall be written after the radix character; if the *precision* is explicitly 0, no radix character shall appear.

`e`,`E` The floating point number argument shall be written in the style `"[−]`*d*`.`*ddd* `e`±*dd*`"` (the symbol ± indicates either a plus or minus sign), where there is one digit before the radix character (shown here as a decimal point) and the number of digits after it is equal to the precision. The LC_NUMERIC locale category shall determine the radix character to use in this format. When the precision is missing, six digits shall be written after the radix character; if the precision is `0`, no radix character shall appear. The `E` conversion character shall produce a number with `E` instead of `e` introducing the exponent. The exponent always shall contain at least two digits. However, if the value to be written requires an exponent greater than two digits, additional exponent digits shall be written as necessary.

`g`,`G` The floating point number argument shall be written in style `f` or `e` (or in style `E` in the case of a `G` conversion character), with the precision specifying the number of significant digits. The style used depends on the value converted: style `e` shall be used only if the exponent resulting from the conversion is less than `−4` or greater than or equal to the precision. Trailing zeroes shall be removed from the result. A radix character shall appear only if it is followed by a digit.

`c` The integer argument shall be converted to an *unsigned char* and the resulting byte shall be written.

`s` The argument shall be taken to be a string and bytes from the string shall be written until the end of the string or the number of bytes indicated by the *precision* specification of the argument is reached. If the precision is omitted from the argument, it shall be taken to be infinite, so all bytes up to the end of the string shall be written.

% Write a % character; no argument shall be converted.

In no case does a nonexistent or insufficient *field width* cause truncation of a field; if the result of a conversion is wider than the field width, the field shall be simply expanded to contain the conversion result.

2.12.1 File Format Notation Rationale. *(This subclause is not a part of P1003.2)*

This clause was originally derived from the description of *printf* () in the *SVID* , but it has been updated following the publication of the C Standard {7}. It is not identical to the C Standard's {7}*printf* (), as it deals with integers as being essentially one type, disregarding possible internal differences between *int* , *short* , and *long* . It has also had some of the internal C language dependencies removed (such as the requirement for null-terminated strings).

This standard provides a rigorous description of the format of utility input and output files. It is the intention of this standard that these descriptions be adequate sources of information so that portable applications can use other utilities such as `lex` or `awk` to reliably parse the output of these utilities as their input in, say a pipeline.

The notation for spaces allows some flexibility for application output. Note that an empty character position in *format* represents one or more <blank> characters on the output (not *white space*, which can include <newline>s). Therefore, another utility that reads that output as its input must be prepared to parse the data using *scanf*(), `awk`, etc. The Δ character is used when exactly one <space> is output.

The treatment of integers and spaces is different from the real *printf*(), in that they can be surrounded with <blank>s. This was done so that, given a format such as:

"%d\n", <*foo*>

the implementation could use a real *printf*() such as

```
printf("%6d\n", foo);
```

and still conform. It would have been possible for the standard to use `"%6d\n"`, but it would have been difficult to pick a number that would have pleased everyone. This notation is thus somewhat like *scanf*() in addition to *printf*().

The *printf*() function was chosen as a model as most of the working group was familiar with it and it was thought that many of the readers would be as well.

One difference from the C function *printf*() is that the `l` and `h` conversion characters are not used. As expressed by this standard, there is no differentiation between decimal values for *int* s versus *long* s versus *short* s. The specifications `%d` or `%i` should be interpreted as an arbitrary length sequence of digits. Also, no distinction is made between single precision and double precision numbers (*float/double* in C). These are simply referred to as floating point numbers.

Many of the output descriptions in this standard use the term *line*, such as:

"%s", <*input line*>

Since the definition of *line* includes the trailing <newline> character already, there is no need to include a `"\n"` in the format; a double <newline> would otherwise result.

In the language at the end of the clause:

> ‘‘In no case does a nonexistent or insufficient *field width* cause truncation of a field; . . . ’’

the term ‘‘field width’’ should not be confused with the term ‘‘precision’’ used in the description of `%s`.

Examples:

To represent the output of a program that prints a date and time in the form `Sunday, July 3, 10:02`, where <*weekday*> and <*month*> are strings:

`"%s,∆%s∆%d,∆%d:%.2d\n"`, *<weekday>* , *<month>* , *<day>*, *<hour>*, *<min>*

To show π written to 5 decimal places:

`"pi∆=∆%.5f\n"`, *<value of* π*>*

To show an input file format consisting of five colon-separated fields:

`"%s:%s:%s:%s:%s\n"`, *<arg1>*, *<arg2>*, *<arg3>*, *<arg4>*, *<arg5>*

2.13 Configuration Values

2.13.1 Symbolic Limits

This clause lists magnitude limitations imposed by a specific implementation. The braces notation, {LIMIT }, is used in this standard to indicate these values, but the braces are not part of the name.

Table 2-16 – Utility Limit Minimum Values

Name	Description	Value	
{POSIX2_BC_BASE_MAX}	The maximum *obase* value allowed by the `bc` utility.	99	
{POSIX2_BC_DIM_MAX}	The maximum number of elements permitted in an array by the `bc` utility.	2048	
{POSIX2_BC_SCALE_MAX}	The maximum *scale* value allowed by the `bc` utility.	99	
{POSIX2_BC_STRING_MAX}	The maximum length of a string constant accepted by the `bc` utility.	1000	
{POSIX2_COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the LC_COLLATE `order` keyword in the locale definition file; see 2.5.2.2.3.	2	
{POSIX2_EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the `expr` utility.	32	
{POSIX2_LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing `<newline>`.	2048	
{POSIX2_RE_DUP_MAX}	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation `\{`*m*`,` *n*`\}`; see 2.8.3.3.	255	
{POSIX2_VERSION }	This value indicates the version of the utilities in this standard that are provided by the implementation. It will change with each published version of this standard.	199???	1

The values specified in Table 2-16 represent the lowest values conforming implementations shall provide; and consequently, the largest values on which an application can rely without further enquiries, as described below. These values shall be accessible to applications via the `getconf` utility (see 4.26) and through the interfaces described in 7.8.2, [such as *sysconf* () in the C binding]. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

Implementations may provide more liberal, or less restrictive, values than shown in Table 2-16. These possibly more liberal values are accessible using the symbols in Table 2-17.

Table 2-17 – Symbolic Utility Limits

Name	Description	Minimum Value
{BC_BASE_MAX}	The maximum *obase* value allowed by the `bc` utility.	{POSIX2_BC_BASE_MAX}
{BC_DIM_MAX}	The maximum number of elements permitted in an array by the `bc` utility.	{POSIX2_BC_DIM_MAX}
{BC_SCALE_MAX}	The maximum *scale* value allowed by the `bc` utility.	{POSIX2_BC_SCALE_MAX}
{BC_STRING_MAX}	The maximum length of a string constant accepted by the `bc` utility.	{POSIX2_BC_STRING_MAX}
{COLL_WEIGHTS_MAX}	The maximum number of weights that can be assigned to an entry of the LC_COLLATE `order` keyword in the locale definition file; see 2.5.2.2.3.	{POSIX2_COLL_WEIGHTS_MAX}
{EXPR_NEST_MAX}	The maximum number of expressions that can be nested within parentheses by the `expr` utility.	{POSIX2_EXPR_NEST_MAX}
{LINE_MAX}	Unless otherwise noted, the maximum length, in bytes, of a utility's input line (either standard input or another file), when the utility is described as processing text files. The length includes room for the trailing `<newline>`.	{POSIX2_LINE_MAX}
{RE_DUP_MAX}	The maximum number of repeated occurrences of a regular expression permitted when using the interval notation `\{`*m*, *n*`\}`; see 2.8.3.3.	{POSIX2_RE_DUP_MAX}

The functions in 7.8.2 [such as *sysconf* () in the C binding] or the `getconf` utility shall return the value of each symbol on each specific implementation. The value so retrieved shall be the largest, or most liberal, value that shall be available throughout the session lifetime, as determined at session creation. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

All numerical limits defined by POSIX.1 {8}, such as {PATH_MAX}, also apply to this standard. (See POSIX.1 {8} 2.8.) All the utilities defined by this standard are implicitly limited by these values, unless otherwise noted in the utility descriptions.

It is not guaranteed that the application can in fact push a value to the implementation's specified limit in any given case, or at all, as a lack of virtual memory or other resources may prevent this. The limit value indicates only that the implementation does not specifically impose any arbitrary, more restrictive limit.

2.13.1.1 Symbolic Limits Rationale. *(This subclause is not a part of P1003.2)*

This clause grew out of an idea that originated in POSIX.1 {8}, in the form of *sysconf*() and *pathconf*(). (In fact, the same person wrote the original text for both standards.) The idea is that a Strictly Conforming POSIX.2 Application can be written to use the most restrictive values that a minimal system can provide, but it shouldn't have to. The values shown in Table 2-17 represent compromises so that some vendors can use historically-limited versions of UNIX system utilities. They are the highest values that Strictly Conforming POSIX.2 Applications or Conforming POSIX.2 Applications can assume, given no other information.

However, by using `getconf` or *sysconf*(), the elegant application can tailor itself to the more liberal values on some of the specific instances of specific implementations.

There is no explicitly-stated requirement that an implementation provide finite limits for any of these numeric values; the implementation is free to provide essentially unbounded capabilities (where it makes sense), stopping only at reasonable points such as {ULONG_MAX} (from the C Standard {7} via POSIX.1 {8}). Therefore, applications desiring to tailor themselves to the values on a particular implementation need to be ready for possibly huge values; it may not be a good idea to blindly allocate a buffer for an input line based on the value of {LINE_MAX}, for instance. However, unlike POSIX.1 {8}, there is no set of limits in this standard that return a special indication meaning ''unbounded.'' The implementation should always return an actual number, even if the number is very large.

The statement

> ''It is not guaranteed that the application ...

is an indication that many of these limits are designed to ensure that implementors design their utilities without arbitrary constraints related to unimaginative programming. There are certainly conditions under which combinations of options can cause failures that would not render an implementation nonconforming. For example, {EXPR_NEST_MAX} and {ARG_MAX} could collide when expressions are large; combinations of {BC_SCALE_MAX} and {BC_DIM_MAX} could exceed virtual memory.

In POSIX.2, the notion of a limit being guaranteed for the process lifetime, as it is in POSIX.1 {8}, is not as useful to a shell script. The `getconf` utility is probably a process itself, so the guarantee would be valueless. Therefore, POSIX.2 requires the guarantee to be for the session lifetime. This will mean that many vendors will either return very conservative values or possibly implement `getconf` as a built-in.

It may seem confusing to have limits that apply only to a single utility grouped into one global clause. However, the alternative, which would be to disperse them out into their utility description clauses, would cause great difficulty when *sysconf*() and `getconf` were described. Therefore, the working group chose the global approach.

Each language binding could provide symbol names that are slightly different than are shown here. For example, the C binding prefixes the symbols with a leading underscore.

The following comments describe selection criteria for the symbols and their values.

{ARG_MAX}
: This is defined by POSIX.1 {8}. Unfortunately, it is very difficult for a portable application to deal with this value, as it does not know how much of its argument space is being consumed by the user's environment variables.

{BC_BASE_MAX}
{BC_DIM_MAX}
{BC_SCALE_MAX}
: These were originally one value, {BC_SCALE_MAX}, but it was unreasonable to link all three concepts into one limit.

{CHILD_MAX}
: This is defined by POSIX.1 {8}.

{CUT_FIELD_MAX}
: This value was removed from an earlier draft. It represented the maximum length of the *list* argument to the `cut -c` or `-f` options. Since the length is now unspecified, the utility should have to deal with arbitrarily long lists, as long as {ARG_MAX} is not exceeded.

{CUT_LINE_MAX}
: This value was removed from an earlier draft. Historical `cut`s have had input line limits of 1024; this removal therefore mandates that a conforming `cut` shall process files with lines of unlimited length. 1

{DEPTH_MAX}
: This directory-traversing depth limit (which at one time applied to `rm` and `find`) was removed from an earlier draft for two major reasons:

(1) It could be a security problem if utilities searching for files could not descend below a published depth; this would be a semi-reliable means of hiding files from the administrator.

(2) There is no reason a reasonable implementation should have to limit itself in this way.

{ED_FILE_MAX}
This value was removed from an earlier draft. Historical `ed`s have had very small file limits; since {ED_FILE_MAX} is no longer specified, implementations have to document the limits as described in 2.11. It is recommended that implementations set much more reasonable file size limits as they modify `ed` to deal with other features required by POSIX.2.

{ED_LINE_MAX}
This value was removed from an earlier draft. Historical `ed`s have had small input line limits; this removal therefore mandates that a conforming `ed` shall process files with lines of length {LINE_MAX}.

{COLL_WEIGHTS_MAX}
The weights assigned to `order` can be considered as ‘‘passes’’ through the collation algorithm.

{EXPR_NEST_MAX}
The value for expression nesting was borrowed from the C Standard {7}.

{FIND_DEPTH_MAX}
This was removed from an earlier draft in favor of a common value, {DEPTH_MAX}.

{FIND_FILESYS_MAX}
This was removed from an earlier draft. It indicated the limit of the number of file systems that `find` could traverse in its search. It was dropped because this standard does not really acknowledge the historical nature of separate file systems.

{FIND_NEWER_MAX}
This value, which allowed `find` to limit the number of `-newer` operands it processed, was deleted from an earlier draft. It was felt to be a vestige of a particular implementation with an incorrect programming algorithm that should not limit applications.

{JOIN_LINE_MAX}
This value was removed from an earlier draft. Historical `join`s have had input line limits of 1024; this removal therefore mandates that a conforming `join` shall process files with lines of length {LINE_MAX}.

{LINE_MAX}
This is a global limit that affects all utilities, unless otherwise noted. The {MAX_CANON} value from POSIX.1 {8} may further limit input lines from terminals. The {LINE_MAX} value was the subject of much debate and is a compromise between those who wished unlimited lines and those who understood that many historical utilities were written with fixed buffers. Frequently, utility writers selected the UNIX system constant BUFSIZ to allocate these buffers; therefore, some utilities were limited to 512 bytes for I/O lines, while others achieved 4096 or greater.

It should be noted that {LINE_MAX} applies only to input line length; there is no requirement in the standard that limits the length of output lines. Utilities such as `awk`, `sed`, and `paste` could theoretically construct lines longer than any of the input lines they received, depending on the options used or the instructions from the application. They are not required to truncate their output to {LINE_MAX}. It is the responsibility of the application to deal with this. If the output of one of those utilities is to be piped into another of the standard utilities, line lengths restrictions will have to be considered; the `fold` utility, among others, could be used to ensure that only reasonable line lengths reach utilities or applications.

{LINK_MAX}
This is defined by POSIX.1 {8}.

{LP_LINE_MAX}
This value was removed from an earlier draft. Since so little is being required for the details of the `lp` utility, it made little sense to specify how long its output lines are. Thus, implementations of `lp` will be expected to deal with lines up to {LINE_MAX}, but whether those lines print sensibly on every device is unspecified.

{MAX_CANON}
This is defined by POSIX.1 {8}.

{MAX_INPUT}
This is defined by POSIX.1 {8}.

{NAME_MAX}
This is defined by POSIX.1 {8}.

{NGROUPS_MAX}
This is defined by POSIX.1 {8}.

{OPEN_MAX}
This is defined by POSIX.1 {8}.

{PATH_MAX}
This is defined by POSIX.1 {8}.

{PIPE_BUF}
This is defined by POSIX.1 {8}.

{RM_DEPTH_MAX}
This was removed from an earlier draft in favor of a common value, {DEPTH_MAX}.

{RE_DUP_MAX}
The value selected is consistent with historical practice.

{SED_PATTERN_MAX}
This symbolic value, the size of the `sed` pattern space, was replaced by a specific value in the `sed` description. It is unlikely that any real application would ever need to access this value symbolically.

{SORT_LINE_MAX}
This was removed from an earlier draft. Now that `cut` and `fold` can handle unlimited-length input lines, a special long input line limit for `sort` is not needed.

There are different limits associated with command lines and input to utilities, depending on the method of invocation. In the case of a C program *exec*-ing a utility, {ARG_MAX} is the underlying limit. In the case of the shell reading a script and *exec*-ing a utility, {LINE_MAX} limits the length of lines the shell is required to process and {ARG_MAX} will still be a limit. If a user is entering a command on a terminal to the shell, requesting that it invoke the utility, {MAX_INPUT} may restrict the length of the line that can be given to the shell to a value below {LINE_MAX}.

2.13.2 Symbolic Constants for Portability Specifications

Table 2-18 – Optional Facility Configuration Values

Name	Description
{POSIX2_C_BIND}	The C language development facilities in Annex A support the C Language Bindings Option (see Annex B).
{POSIX2_C_DEV}	The system supports the C Language Development Utilities Option (see Annex A).
{POSIX2_FORT_DEV}	The system supports the FORTRAN Development Utilities Option (see Annex C).
{POSIX2_FORT_RUN}	The system supports the FORTRAN Runtime Utilities Option (see Annex C).
{POSIX2_LOCALEDEF}	The system supports the creation of locales as described in 4.35.
{POSIX2_SW_DEV}	The system supports the Software Development Utilities Option (see Section 6).

Table 2-18 lists symbols that can be used by the application to determine which optional facilities are present on the implementation. The functions defined in 7.8.2 [such as *sysconf*()] or the `getconf` utility can be used to retrieve the value of each symbol on each specific implementation. The literal names shown in the table apply only to the `getconf` utility; the high-level-language binding shall describe the exact form of each name to be used by the interfaces in that binding.

Each of these symbols shall be considered valid names by the implementation. Each shall be defined on the system with a value of 1 if the corresponding option is supported; otherwise, the symbol shall be undefined.

2.13.2.1 Symbolic Constants for Portability Specifications Rationale. *(This subclause is not a part of P1003.2)*

When an option is supported, `getconf` returns a value of 1. For example, when C development is supported:

```
if [ "$(getconf POSIX2_C_DEV)" -eq 1 ]; then
        echo C supported
fi
```

The *sysconf* () function in the C binding would return 1.

The following comments describe selection criteria for the symbols and their values.

{POSIX 2_C_BIND }
{POSIX 2_C_DEV }
{POSIX 2_FORT _DEV }
{POSIX 2_SW_DEV }

These were renamed from _POSIX _* in Draft 9 after it was pointed out that each of the POSIX standards should keep generally in its own namespace.

It is possible for some (usually privileged) operations to remove utilities that support these options, or otherwise render these options unsupported. The header files, the *sysconf* () function, or the `getconf` utility will not necessarily detect such actions, in which case they should not be considered as rendering the implementation nonconforming. A test suite should not attempt tests like:

```
rm /usr/bin/c89
getconf POSIX2_C_DEV
```

{_POSIX _LOCALEDEF }

This symbol was introduced to allow implementations to restrict supported locales to only those supplied by the implementation.

Section 3: Shell Command Language

The shell is a command language interpreter. This section describes the syntax of that command language as it is used by the `sh` utility and the functions in 7.1 [such as *system* () and *popen* () in the C binding].

The shell operates according to the following general overview of operations. The specific details are included in the cited clauses and subclauses of this section. The shell:

(1) Reads its input from a file (see `sh` in 4.56), from the `−c` option, or from one of the functions in 7.1. If the first line of a file of shell commands starts with the characters #!, the results are unspecified.

(2) Breaks the input into tokens: words and operators. (See 3.3.)

(3) Parses the input into simple (3.9.1) and compound (3.9.4) commands.

(4) 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 (3.6).

(5) Performs redirection (3.7) and removes redirection operators and their operands from the parameter list.

(6) Executes a function (3.9.5), built-in (3.14), executable file, or script, giv-
ing the name of the command (or, in the case of a function within a 1
script, the name of the script) as the ''zero'th'' argument and the remain- 1
ing words and fields as parameters (3.9.1.1).

(7) Optionally waits for the command to complete and collects the exit status (3.8.2).

3.0.1 Shell Command Language Rationale. *(This subclause is not a part of P1003.2)*

The System V shell was selected as the starting point for this standard. The BSD C-shell was excluded from consideration, for the following reasons:

(1) Most historically portable shell scripts assume the Version 7 ''Bourne'' shell, from which the System V shell is derived.

(2) The majority of tutorial materials on shell programming assume the System V shell.

Despite the selection of the System V shell, the developers of the standard did not limit the possibilities for a shell command language that was upward-compatible.

The only programmatic interfaces to the shell language are through the functions in 7.1 and the `sh` utility. Most implementations provide an interface to, and processing mode for, the shell that is suitable for direct user interaction. The behavior of this interactive mode is not defined by this standard; however, places where historically an interactive shell behaves differently from the behavior described here are noted.

(1) Aliases are not included in the base POSIX.2 because they duplicate functionality already available to applications with functions. In early drafts, the search order of simple command lookup was ''aliases, built-ins, functions, file system,'' and therefore an alias was necessary to create a user-defined command having the same name as a built-in. To retain this capability, the search order has changed to ''special built-ins, functions, built-ins, file system,'' and a built-in, called `command`, has been added, which disables the looking up of functions. Aliases are a part of the POSIX.2a UPE because they are widely used by human users, as differentiated from applications.

(2) All references to job control and related commands have been omitted from the base POSIX.2. POSIX.2 describes the noninteractive operation of the shell; job control is outside the scope of this standard until the UPE revision is developed. Apparently it is not widely known that traditionally, even in a job control environment, the commands executed during the execution of a shell script are not placed into separate process groups. If they were, one could not stop the execution of the shell script from the interactive shell, for example. This standard does not require or prohibit job control; it simply does not mention it.

(3) The conditional command (double bracket `[[ ]]`) was removed from an earlier draft. Objections were lodged that the real problem is misuse of the `test` command (`[`), and putting it into the shell is the wrong way to fix the problem. Instead, proper documentation and a new shell reserved word (`!`) are sufficient. Tests that require multiple `test` operations can be done at the shell level using individual invocations of the `test` command and shell logicals, rather than the error prone `-o` flag of `test`.

(4) Exportable functions were removed from an earlier draft. See the rationale in 3.9.5.1.

The construct #! is reserved for implementations wishing to provide that extension. If it were not reserved, the standard would disallow it by forcing it to be a comment. As it stands, a conforming application shall not use #! as the first line of a shell script.

3.1 Shell Definitions

The following terms are used in Section 3. Because they are specific to the shell, they do not appear in 2.2.2.

3.1.1 control operator: A token that performs a control function.

It is one of the following symbols:

`&`	`)`	`<newline>`
`&&`	`;`	`|`
`(`	`;;`	`||`

The end-of-input indicator used internally by the shell is also considered a control operator. See 3.3.

On some systems, the symbol `((` is a control operator; its use produces unspecified results. 1

3.1.2 expand: When not qualified, the act of applying all the expansions described in 3.6.

3.1.3 field: A unit of text that is the result of parameter expansion (3.6.2), arithmetic expansion (3.6.4), command substitution (3.6.3), or field splitting (3.6.5).

During command processing (see 3.9.1), the resulting fields are used as the command name and its arguments.

3.1.4 interactive shell: A processing mode of the shell that is suitable for direct user interaction.

The behavior in this mode is not defined by this standard.

NOTE: The preceding sentence is expected to change following the eventual approval of the UPE supplement.

3.1.5 name: A word consisting solely of underscores, digits, and alphabetics from the portable character set (see 2.4).

The first character of a name shall not be a digit.

3.1.6 operator: Either a control operator or a redirection operator.

3.1.7 parameter: An entity that stores values.

There are three types of parameters: variables (named parameters), positional parameters, and special parameters. Parameter expansion is accomplished by introducing a parameter with the `$` character. See 3.5.

3.1.8 positional parameter: A parameter denoted by a single digit or one or more digits in curly braces.

See 3.5.1.

3.1.9 redirection: A method of associating files with the input/output of commands.

See 3.7.

3.1.10 redirection operator: A token that performs a redirection function.

It is one of the following symbols:

< > >| << >> <& >& <<- <>

3.1.11 special parameter: A parameter named by a single character from the following list:

* @ # ? ! – $ 0

See 3.5.2.

3.1.12 subshell: A shell execution environment, distinguished from the main or current shell execution environment by the attributes described in 3.12.

3.1.13 token: A sequence of characters that the shell considers as a single unit when reading input, according to the rules in 3.3.

A token is either an operator or a word.

3.1.14 variable: A named parameter. See 3.5.

3.1.15 variable assignment [assignment]: A word consisting of the following parts

varname =*value*

When used in a context where assignment is defined to occur (see 3.9.1) and at no other time, the *value* (representing a word or field) shall be assigned as the value of the variable denoted by *varname* . The *varname* and *value* parts meet the requirements for a name and a word, respectively, except that they are delimited by the embedded unquoted equals-sign in addition to the delimiting described in 3.3. In all cases, the variable shall be created if it did not already exist. If *value* is not specified, the variable shall be given a null value.

An alternative form of variable assignment:

symbol =*value*

(where *symbol* is a valid word delimited by an equals-sign, but not a valid name) produces unspecified results.

3.1.16 word: A token other than an operator.

In some cases a word is also a portion of a word token: in the various forms of parameter expansion (3.6.2), such as ${*name* –*word* }, and variable assignment, such as *name*=*word* , the word is the portion of the token depicted by *word* . The concept of a word is no longer applicable following word expansions—only fields remain; see 3.6.

3.1.17 Shell Definitions Rationale. *(This subclause is not a part of P1003.2)*

The *word*=*word* form of variable assignment was included, producing unspecified results, to allow the KornShell *name* [*expression*]=*value* syntax to conform.

The `((` symbol is a control operator in the KornShell, used for an alternative syntax of an arithmetic expression command. A strictly conforming POSIX.2 application cannot use `((` as a single token [with the obvious exception of the `$((` form described in POSIX.2]. The decision to require this is based solely on the pragmatic knowledge that there are many more historical shell scripts using the KornShell syntax than there might be using nested subshells, such as

```
((foo))        or        ((foo);(bar))
```

The latter example should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. Thus, in most cases, while a few scripts will no longer be strictly portable, the chances of breaking existing scripts is even smaller.

There are no explicit limits in this standard on the sizes of names, words, lines, or other objects. However, other implicit limits do apply: shell script lines produced by many of the standard utilities cannot exceed {LINE_MAX} and the sum of exported variables comes under the {ARG_MAX} limit. Historical shells dynamically allocate memory for names and words and parse incoming lines a byte at a time. Lines cannot have an arbitrary {LINE_MAX} limit because of historical practice such as makefiles, where `make` removes the `<newline>`s associated with the commands for a target and presents the shell with one very long line. The text in 2.11.5.2 does allow a shell to run out of memory, but it cannot have arbitrary programming limits.

3.2 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 recognized as such; and prevent parameter expansion and command substitution within here-document processing (see 3.7.4).

The following characters shall 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 the standard:

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

The various quoting mechanisms are the escape character, single-quotes, and double-quotes. The here-document represents another form of quoting; see 3.7.4.

3.2.1 Escape Character (Backslash)

A backslash that is not quoted shall preserve the literal value of the following character, with the exception of a <newline>. If a <newline> follows the backslash, the shell shall interpret this as line continuation. The backslash and <newline> shall be removed before splitting the input into tokens.

3.2.2 Single-Quotes

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

3.2.3 Double-Quotes

Enclosing characters in double-quotes (" ") shall preserve 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 shall retain its special meaning introducing parameter expansion (see 3.6.2), a form of command substitution (see 3.6.3), and arithmetic expansion (see 3.6.4).

The input characters within the quoted string that are also enclosed between $(and the matching) shall not be affected by the double-quotes, but rather shall define that command whose output replaces the $(...) when the word is expanded. The tokenizing rules in 3.3 shall be 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, shall occur. A preceding backslash character shall be used to escape a literal { or }. The rule in 3.6.2 shall be used to determine the matching }.

` The backquote shall retain its special meaning introducing the other form of command substitution (see 3.6.3). 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 shall retain its special meaning as an escape character (see 3.2.1) only when followed by one of the characters:

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

A double-quote shall be preceded by a backslash to be included within double-quotes. The parameter @ has special meaning inside double-quotes and is described in 3.5.2.

3.2.4 Quotes Rationale. *(This subclause is not a part of P1003.2)*

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 3.6.5 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/escaping, thus permitting any combination of characters.

The escaped `<newline>` used for line continuation is removed entirely from the input and is not replaced by any white space. Therefore, it cannot serve as a token separator.

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
```

It would be desirable to include the statement ‘‘The characters from an enclosed `${` to the matching `}` shall not be affected by the double-quotes,’’ similar to the one for `$( )`. However, historical practice in the System V shell prevents this. The requirement that double-quotes be matched inside `${...}` within double-quotes and the rule for finding the matching `}` in 3.6.2 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 `"${...}"` form have led to inconsistencies between the historical System V, BSD, and KornShells, and the text in POSIX.2 is an attempt to converge them without breaking many applications. A consequence of the new rule is that single-quotes cannot be used to quote the `}` within `"${...}"`; 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-\}}"
```

The only alternative to this compromise between shells would be to make the behavior unspecified whenever the literal characters ', {, }, and " appear within `${...}`. To write a portable script that uses these values, a user would have to assign variables, say,

```
squote=\' dquote=\" lbrace='{' rbrace='}'
${foo-$squote$rbrace$squote}
```

rather than

```
${foo-"'}'"}
```

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

```
"`echo hello"
```

This usage is undefined in POSIX.2, where the matching backquote is required. 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 recognizing 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).

3.3 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 1
input lines can be of unlimited length. These lines are parsed using two major 1
modes: ordinary token recognition and processing of here-documents.

When an `io_here` token has been recognized by the grammar (see 3.10), one or more of the immediately subsequent lines form the body of one or more here-documents and shall be parsed according to the rules of 3.7.4.

When it is not processing an io_here, the shell shall break its input into tokens 1 by applying the first applicable rule below to the next character in its input. The token shall 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 shall continue until an actual token is delimited.

(1) If the end of input is recognized, the current token shall be delimited. If there is no current token, the end-of-input indicator shall 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 shall be used as part of that (operator) token.

(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 shall be delimited.

(4) If the current character is backslash, single-quote, or double-quote (\, ', or ") and it is not quoted, it shall affect quoting for subsequent character(s) up to the end of the quoted text. The rules for quoting are as described in 3.2. During token recognition no substitutions shall be actually performed, and the result token shall contain exactly the characters that appear in the input (except for <newline> joining), unmodified, including any embedded or enclosing quotes or substitution operators, between the quote mark and the end of the quoted text. The token shall not be delimited by the end of the quoted field.

(5) If the current character is an unquoted $ or `, the shell shall identify the start of any candidates for parameter expansion (3.6.2), command substitution (3.6.3), or arithmetic expansion (3.6.4) from their introductory unquoted character sequences: $ or ${, $( or `, and $((, respectively. The shell shall read sufficient input to determine the end of the unit to be expanded (as explained in the cited subclauses). While processing the characters, if instances of expansions or quoting are found nested within the substitution, the shell shall 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 recognize embedded constructs, shall be included unmodified in the result token, including any embedded or enclosing substitution operators or quotes. The token shall 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) shall be delimited. The current character shall be used as the beginning of the next (operator) token.

(7) If the current character is an unquoted `<newline>`, the current token shall be delimited.

(8) If the current character is an unquoted `<blank>`, any token containing the previous character is delimited and the current character is discarded.

(9) If the previous character was part of a word, the current character is appended to that word.

(10) If the current character is a #, it and all subsequent characters up to, but excluding, the next `<newline>` are discarded as a comment. The `<newline>` that ends the line is not considered part of the comment.

(11) The current character is used as the start of a new word.

Once a token is delimited, it shall be categorized as required by the grammar in 3.10.

3.3.1 Token Recognition Rationale. *(This subclause is not a part of P1003.2)*

The (3) rule about combining characters to form operators is not meant to preclude systems from extending the shell language when characters are combined in otherwise invalid ways. Portable applications cannot use invalid combinations and test suites should not penalize systems that take advantage of this fact. For example, the unquoted combination |& is not valid in a POSIX.2 script, but has a specific KornShell meaning.

The (10) rule about # as the current character is the first in the sequence in which a new token is being assembled. The # starts a comment only when it is at the beginning of a token. This rule is also written to indicate that the search for the end-of-comment does not consider escaped `<newline>` specially, so that a comment cannot be continued to the next line.

3.4 Reserved Words

Reserved words are words that have special meaning to the shell. (See 3.9.) The following words shall be recognized as reserved words:

```
!       elif    fi      in      while
case    else    for     then    {
do      esac    if      until   }
done
```
[4]

4) In some historical systems, the curly braces are treated as control operators. To assist in future standardization activities, portable applications should avoid using unquoted braces to represent the characters themselves. It is possible that a future version of POSIX.2 may require this, although probably not for the often-used `find {}` construct.

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

(1) The first word of a command

(2) The first word following one of the reserved words other than `case`, `for`, or `in`

(3) The third word in a `case` or `for` command (only `in` is valid in this case)

See the grammar in 3.10.

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

`function` `select` `[[` `]]` 2

Words that are the concatenation of a name and a colon (`:`) are reserved; their use produces unspecified results.

3.4.1 Reserved Words Rationale. *(This subclause is not a part of P1003.2)*

All reserved words are recognized syntactically as such in the contexts described. However, it is useful to point out that `in` is the only meaningful reserved word after a `case` or `for`; similarly, `in` is not meaningful as the first word of a simple command.

Reserved words are recognized only when they are delimited (i.e., meet the definition of *word* ; see 3.1.16), whereas operators are themselves delimiters. For instance, `(` and `)` are control operators, so that no `<space>` is needed in `(list)`. However, `{` and `}` are reserved words in `{ list;}`, so that in this case the leading `<space>` and semicolon are required.

The list of unspecified reserved words is from the KornShell, so portable applica-
tions cannot use them in places a reserved word would be recognized. This list 2
contained `time` in earlier drafts, but it was removed when the `time` utility was 2
selected for the UPE. 2

There was a strong argument for promoting braces to operators (instead of reserved words), so they would be syntactically equivalent to subshell operators. Concerns about compatibility outweighed the advantages of this approach. Nevertheless, portable applications should consider quoting `{` and `}` when they represent themselves.

The restriction on ending a name with a colon is to allow future implementations that support named labels for flow control. See the rationale for `break` (3.14.1.1).

3.5 Parameters and Variables

A parameter can be denoted by a name, a number, or one of the special characters listed in 3.5.2. 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.

3.5.1 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. When a positional parameter with more than one digit is specified, the application shall enclose the digits in braces (see 3.6.2). Positional parameters are initially assigned when the shell is invoked (see `sh` in 4.56), temporarily replaced when a shell function is invoked (see 3.9.5), and can be reassigned with the `set` special built-in command.

3.5.1.1 Positional Parameters Rationale. *(This subclause is not a part of P1003.2)*

The digits denoting the positional parameters are always interpreted as a decimal value, even if there is a leading zero.

3.5.2 Special Parameters

Listed below are the special parameters and the values to which they shall expand. Only the values of the special parameters are listed; see 3.6 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 3.2.3), 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>` if **IFS** is unset.

@ Expands to the positional parameters, starting from one. When the expansion occurs within double-quotes, 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 1
expansion of @ shall generate zero fields, even when @ is double-quoted. 1

Expands to the decimal number of positional parameters.

? Expands to the decimal exit status of the most recent pipeline (see 3.9.2).

– (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.

`$` Expands to the decimal process ID of the invoked shell. In a subshell (see 3.12), `$` shall expand to the same value as that of the current shell.

`!` Expands to the decimal process ID of the most recent background command (see 3.9.3) executed from the current shell. 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` (4.56) for a detailed description of how this name is derived.

See the description of the **IFS** variable in 3.5.3.

3.5.2.1 Special Parameters Rationale. *(This subclause is not a part of P1003.2)*

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

If a subshell were to execute a background command, the value of its parent's `$!` would not change. For example:

```
(
date &
echo $!
)
echo $!
```

would echo two different values for `$!`.

The descriptions of parameters * and @ assume the reader is familiar with the field splitting discussion in 3.6.5 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.

Historical versions of the Bourne shell have used `<space>` as a separator between the expanded members of `"$*"`. The KornShell has used the first character in **IFS**, which is `<space>` by default. 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
```

The `$-` can be used to save and restore `set` options:

```
Save=$(echo $- | sed 's/[ics]//g')
...
set +aCefnuvx
set -$Save
```

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`.

The command name (parameter `0`) is not counted in the number given by # because it is a special parameter, not a positional parameter.

3.5.3 Variables

Variables shall be initialized from the environment (as defined by POSIX.1 {8}) and can be given new values with variable assignment commands. If a variable is initialized from the environment, it shall be marked for export immediately; see 3.14.8. New variables can be defined and initialized with variable assignments, with the `read` or `getopts` utilities, with the *name* parameter in a `for` loop (see 3.9.4.2), with the `${`*name*`=`*word*`}` expansion, or with other mechanisms provided as implementation extensions. The following variables shall affect the execution of the shell:

HOME This variable shall be interpreted as the pathname of the user's home directory. The contents of **HOME** are used in Tilde Expansion (see 3.6.1).

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 shall behave as if the value of **IFS** were the `<space>`, `<tab>`, and `<newline>` characters. (See 3.6.5.)

LANG This variable shall provide a default value for the **LC_*** variables, as described in 2.6.

LC_ALL This variable shall interact with the **LANG** and **LC_*** variables as described in 2.6.

LC_COLLATE This variable shall determine the behavior of range expressions, equivalence classes, and multicharacter collating elements within pattern matching.

LC_CTYPE This variable shall determine the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters), which characters are defined as letters (character class `alpha`), and the behavior of character classes within pattern matching.

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

PATH This variable represents a string formatted as described in 2.6, used to effect command interpretation. See 3.9.1.1.

3.5.3.1 Variables Rationale. *(This subclause is not a part of P1003.2)*

A description of **PWD** (which is automatically set by the KornShell whenever the current working directory changes) was omitted because its functionality is easily reproduced using `$(pwd)`.

See the discussion of **IFS** in 3.6.5.1.

Other common environment variables used by historical shells are not specified by this standard, but they should be reserved for the historical uses. For interactive use, other shell variables are expected to be introduced by the UPE (and this rationale will be updated accordingly): **ENV**, **FCEDIT**, **HISTFILE**, **HISTSIZE**, **LINENO**, **PPID**, **PS 1**, **PS 2**, **PS 4**.

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 3.6.1. Note that the tildes are expanded during the assignment to **PATH**, not when **PATH** is accessed during command search.

3.6 Word Expansions

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

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 is described in 3.5.2.

The order of word expansion shall be as follows:

(1) Tilde Expansion (see 3.6.1), Parameter Expansion (see 3.6.2), Command Substitution (see 3.6.3), and Arithmetic Expansion (see 3.6.4) shall be performed, beginning to end. [See item (5) in 3.3.]

(2) Field Splitting (see 3.6.5) shall be performed on fields generated by step (1) unless **IFS** is null.

(3) Pathname Expansion (see 3.6.6) shall be performed, unless `set –f` is in effect.

(4) Quote Removal (see 3.6.7) shall always be performed last.

The expansions described in this clause shall 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 shall 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 3.5.2), a valid first character of a variable name, a left curly brace (`{`), or a left parenthesis, the result is unspecified.

3.6.0.1 Word Expansions Rationale. *(This subclause is not a part of P1003.2)*

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 behavior. All useful applications of **IFS** use it for parsing input of the `read` utility and for splitting the results of parameter and command substitution. New versions of the shell have fixed this bug, and POSIX. 2 requires the corrected behavior.

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 (1) indicates that Tilde Expansion, Parameter Expansion, Command Substi- 1
tution, and Arithmetic Expansion are all processed simultaneously as they are
scanned. For example, the following is valid arithmetic:

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

An earlier draft stated that Tilde Expansion preceded the other steps, but this is 1
not the case in known historical implementations; if it were, and a referenced 1
home directory contained a $ character, expansions would result within the direc- 1
tory name. 1

3.6.1 Tilde Expansion

A *tilde-prefix* consists of an unquoted tilde character at the beginning of a word, 2
followed by all of the characters preceding the first unquoted slash in the word, or 2
all the characters in the word if there is no slash. In an assignment (see 3.1.15), 2
multiple tilde prefixes can be used: at the beginning of the word (i.e., following 2
the equals-sign of the assignment) and/or following any unquoted colon. A tilde 2
prefix in an assignment is terminated by the first unquoted colon or slash. If none 2
of the characters in the tilde-prefix are quoted, the characters in the tilde-prefix 1
following the tilde shall be treated as a possible login name from the user data-
base (see POSIX.1 {8} Section 9). A portable login name cannot contain characters 2
outside the set given in the description of the **LOGNAME** environment variable in 2
POSIX.1 {8}. If the login name is null (i.e., the tilde-prefix contains only the tilde),
the tilde-prefix shall be replaced by the value of the variable **HOME**. If **HOME** is
unset, the results are unspecified. Otherwise, the tilde-prefix shall be replaced by
a pathname of the home directory associated with the login name obtained using
the equivalent of the POSIX.1 {8} *getpwnam*() function. If the system does not 1
recognize the login name, the results are undefined.

3.6.1.1 Tilde Expansion Rationale. *(This subclause is not a part of P1003.2)*

The text about quoting of the word indicates that `\~hlj/`, `~h\lj/`, `~"hlj"/`, `~hlj\/`, and `~hlj/` are not equivalent: only the last will cause tilde expansion.

Tilde expansion generally occurs only at the beginning of words, but POSIX.2 has adopted an exception based on historical practice in the KornShell:

```
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 behavior because any of the following are reasonable substitutes:

```
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
```

(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.)

The exception was included to avoid breaking numerous KornShell scripts and interactive users and despite the fact that variable assignments in scripts derived from other systems will have to use quoting in some cases to allow literal tildes in strings. (This latter problem should be relatively rare because only tildes preceding known login names in unquoted strings are affected.)

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).

In an earlier draft, tilde expansion occurred following any unquoted equals-sign or colon, but this was removed because of its complexity and to avoid breaking commands such as:

```
rcp hostname:~marc/.profile .
```

A suggestion was made that the special sequence '`$~`'' should be allowed to force tilde expansion anywhere. Since this is not historical practice, it has been left for future implementations to evaluate. (The description in 3.2 requires that a dollar-sign be quoted to represent itself, so the `$~` combination is already unspecified.)

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.

3.6.2 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, shall not be examined in determining the matching `}`.

The simplest form for parameter expansion is:

`${`*parameter*`}`

The value, if any, of *parameter* shall 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 shall 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 shall use the longest valid name (see 3.1.5), whether or not the symbol represented by that name exists. If a parameter expansion occurs inside double-quotes:

— Pathname expansion shall not be performed on the results of the expansion.

— Field splitting shall not be performed on the results of the expansion, with the exception of `@`; see 3.5.2.

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* shall be subjected to tilde expansion, parameter expansion, command substitution, and arithmetic expansion. If *word* is not needed, it shall not be expanded. The `}` character that delimits the following parameter expansion modifications shall be determined as described previously in this subclause and in 3.2.3. (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* shall be substituted; otherwise, the value of *parameter* shall be substituted.

`${`*parameter*`:=`*word*`}` **Assign Default Values.** If *parameter* is unset or null, the expansion of *word* shall be assigned to *parameter*. In all cases, the final value of *parameter* shall 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) shall be written to standard error and the shell shall exit with a nonzero exit status. Otherwise, the value of *parameter* shall be substituted. An interactive shell need not exit.

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

`${#`*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 3.13), rather than regular expression notation, shall 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 shall not cause the following four varieties of pattern char- 1
acters to be quoted, whereas quoting characters within the braces shall have this effect.

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

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

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

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

3.6.2.1 Parameter Expansion Rationale. *(This subclause is not a part of P1003.2)*

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.

The rule for finding the closing `}` in `${...}` is the one used in the KornShell and is upward compatible with the Bourne shell, which does not determine the closing `}` until the word is expanded. The advantage of this is that incomplete expansions, such as

```
${foo
```

can be determined during tokenization, rather than during expansion.

The four expansions with the optional colon have been hard to understand from the historical documentation. The following table summarizes 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 *word*	
`${`*parameter* `:?`*word* `}`	substitute *parameter*	error, exit	error, exit	
`${`*parameter* `?`*word* `}`	substitute *parameter*	substitute null	error, exit	
`${`*parameter* `:+`*word* `}`	substitute *word*	substitute null	substitute null	1
`${`*parameter* `+`*word* `}`	substitute *word*	substitute *word*	substitute null	1

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.

The string length and substring capabilities were included because of the demonstrated need for them, based on their usage in other shells, such as C-shell and KornShell.

Historical versions of the KornShell have not performed tilde expansion on the word part of parameter expansion; however, it is more consistent to do so.

Examples

`${`*parameter* `:–`*word* `}`

In this example, `ls` is executed only if `x` is null or unset. [The `$(ls)` command substitution notation is explained in 3.6.3.]

```
${x:-$(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.

3.6.3 Command Substitution

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

`$(`*command* `)`

or (‘‘backquoted’’ version):

`‘`*command* `‘`

The shell shall expand the command substitution by executing *command* in a subshell environment (see 3.12) 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>`s at the end of the substitution. (Embedded `<newline>`s before the end of the output shall not be removed; however, during field splitting, they may be translated into `<space>`s, 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 shall be satisfied by the first backquote found without a preceding backslash; during this search, if a nonescaped 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 shall not be processed for further tilde expansion, parameter expansion, command substitution, or arithmetic expansion. If a command substitution occurs inside double-quotes, field splitting and pathname expansion shall not be performed on the results of the substitution.

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

    \`*command* \`

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

    $((*command*))

a conforming application shall separate the `$(` and `(` into two tokens (i.e., separate them with white space).

3.6.3.1 Command Substitution Rationale. *(This subclause is not a part of P1003.2)*

The new `$( )` form of command substitution was adopted from the KornShell to solve a problem of inconsistent behavior 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 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 ')'
`                               )
```

Some historical KornShell implementations did not process the first two examples correctly, but the author has agreed to make the appropriate modifications to do so. The KornShell will also be modified so that the following works:

```
echo $(
        case word in
                [Ff]oo) echo found foo ;;
        esac
)
```

Because of these inconsistent behaviors, the backquoted variety of command substitution is not recommended for new applications that nest command substitutions or attempt to embed complex scripts. Because of its widespread historical use, particularly by interactive users, however, the backquotes were retained in POSIX.2 without being declared obsolescent.

The KornShell feature:

> If *command* is of the form <*word* , *word* is expanded to generate a pathname, and the value of the command substitution is the contents of this file with any trailing <newline>s deleted.

was omitted from this standard because `$(cat word)` is an appropriate substitute. However, to prevent breaking numerous scripts relying on this feature, it is unspecified to have a script within `$( )` that has only redirections.

The requirement to separate `$(` and `(` when a single subshell is command-substituted is to avoid any ambiguities with Arithmetic Expansion. See 3.6.4.1.

3.6.4 Arithmetic Expansion

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

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

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

Next, the shell shall treat this as an arithmetic expression and substitute the value of the expression. The arithmetic expression shall be processed according to the rules given in 2.9.2.1, with the following exceptions:

(1) Only integer arithmetic is required.

(2) The `sizeof()` operator and the prefix and postfix `++` and `--` operators are not required.

(3) Selection, Iteration, and Jump Statements are not supported.

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

3.6.4.1 Arithmetic Expansion Rationale. *(This subclause is not a part of P1003.2)*

Numerous ballots were received objecting to the inclusion of the `(( ))` form of KornShell arithmetic in previous drafts. The developers of the standard concluded that there is a strong desire for some kind of arithmetic evaluator to replace `expr`, and that tying it in with `$` makes it fit in nicely with the standard shell language, and provides access to arithmetic evaluation in places where accessing a utility would be inconvenient or clumsy.

Following long debate by interested members of the balloting group, the syntax and semantics for arithmetic were changed. The language is essentially a pure arithmetic evaluator of constants and operators (excluding assignment) and represents a simple subset of the previous arithmetic language [which was derived from the KornShell's `(( ))` construct]. The syntax was changed from that of a command denoted by `((`*expression*`))`, to an expansion denoted by `$((`*expression*`))`. The new form is a dollar expansion (`$`), which evaluates the expression and substitutes the resulting value. Objections to the previous style of arithmetic included that it was too complicated, did not fit in well with the shell's use of variables, and the syntax conflicted with subshells. The justification for the new syntax is that the shell is traditionally a macro language, and if a new feature is to be added, it should be done by extending the capabilities presented by the current model of the shell, rather than by inventing a new one outside the model: adding a new dollar expansion was perceived to be the most intuitive and least destructive way to add such a new capability.

In Drafts 9 and 10, a form `$[`*expression*`]` was used. It was functionally equivalent to the `$(( ))` of the current text, but objections were lodged that the 1988 KornShell had already implemented `$(( ))` and there was no compelling reason to invent yet another syntax. Furthermore, the `$[]` syntax had a minor incompatibility involving the patterns in `case` statements.

The portion of the C Standard {7} arithmetic operations selected corresponds to the operations historically supported in the KornShell.

A simple example using arithmetic expansion:

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

It was concluded that the `test` command (`[`) was sufficient for the majority of relational arithmetic tests, and that tests involving complicated relational expressions within the shell are rare, yet could still be accommodated by testing the value of `$(())` itself. For example:

```
# a complicated relational expression
while [ $(( (($x + $y)/($a * $b)) < ($foo*$bar) )) −ne 0 ]
```

or better yet, the rare script that has many complex relational expressions could define a function like this:

```
val() {
        return $((!$1))
}
```

and complicated tests would be less intimidating:

```
while val $(( (($x + $y)/($a * $b)) < ($foo*$bar) ))
do
        # some calculations
done
```

Another suggestion was to modify `true` and `false` to take an optional argument, and `true` would exit true only if the argument is nonzero, and `false` would exit false only if the argument is nonzero. The suggestion was not favorably received by the balloting group (those contacted were negative about it, all others were silent in their latest ballots).

```
while true $(($x > 5 && $y <= 25))
```

There is a minor portability concern with the new syntax. The example `$((2+2))` could have been intended to mean a command substitution of a utility named `2+2` in a subshell. The developers of POSIX.2 consider this to be obscure and isolated to some KornShell scripts [because `$( )` command substitution existed previously only in the KornShell]. The text on Command Substitution has been changed to require that the `$(` and `(` be separate tokens if this usage is needed.

An example such as

```
echo $((echo hi);(echo there))
```

should not be misinterpreted by the shell as arithmetic because attempts to balance the parentheses pairs would indicate that they are subshells. However, as indicated by 3.1.1, a conforming application must separate two adjacent parentheses with white space to indicate nested subshells.

3.6.5 Field Splitting

After parameter expansion (3.6.2), command substitution (3.6.3), and arithmetic expansion (3.6.4) the shell shall scan the results of expansions and substitutions that did not occur in double-quotes for field splitting and multiple fields can result.

The shell shall 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 `<space>`, `<tab>`, and `<newline>`, or if it is unset, any sequence of `<space>`, `<tab>`, or `<newline>` characters at the beginning or end of the input shall be ignored and any sequence of those characters within the input shall 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 shall be performed.

(3) Otherwise, the following rules shall 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 (e.g., if **IFS** contains <space><comma><tab>, any sequence of <space> and <tab> characters is considered **IFS** white space).

(a) **IFS** white space shall be 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, shall delimit a field, as described previously.

(c) Nonzero-length **IFS** white space shall delimit a field.

3.6.5.1 Field Splitting Rationale. *(This subclause is not a part of P1003.2)*

The operation of field splitting using **IFS** as described in earlier drafts was based on the way the KornShell splits words, but is incompatible with other common versions of the shell. However, each has merit, and so a decision was made to allow both. If the **IFS** variable is unset, or is <space><tab><newline>, the operation is equivalent to the way the System V shell splits words. Using characters outside the <space><tab><newline> set yields the KornShell behavior, where each of the non-<space><tab><newline> characters is significant. This behavior, which affords the most flexibility, was taken from the way the original awk handled field splitting.

The (3) rule can be summarized 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>, the string

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

yields the three colors as the delimited fields.

3.6.6 Pathname Expansion

After field splitting, if set −f is not in effect, each field in the resulting command line shall be expanded using the algorithm described in 3.13, qualified by the rules in 3.13.3.

3.6.7 Quote Removal

The quote characters

\ ' "

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

3.7 Redirection

Redirection is used to open and close files for the current shell execution environment (see 3.12) or for any command. *Redirection operators* can be used with numbers representing file descriptors (see the definition in POSIX.1 {8}) as described below. See also 2.9.1. The relationship between these file descriptors and access to them in a programming language is specified in the language binding for that language to this standard.

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 shall be delimited from any preceding text and immediately precede the redirection operator *redir-op*. If *n* is quoted, the number shall not be recognized 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 shall be recognized. (For example, `echo 2\>a` writes the characters `2>a` to standard output.) The optional number, redirection operator, and *word* shall not appear in the arguments provided to the command to be executed (if any).

In this standard, open files are represented by decimal numbers starting with zero. It is implementation defined what the largest value can be; however, all implementations shall support at least 0 through 9 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 to standard error. The redirection operators can be preceded by one or more digits (with no intervening <blank>s allowed) to designate the file descriptor number.

If the redirection operator is `<<` or `<<-`, the word that follows the redirection operator shall 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 shall be subjected to tilde expansion, parameter expansion, command substitution, arithmetic expansion, and quote removal. Pathname expansion shall not be performed on the word by a noninteractive shell; an interactive shell may perform it, but shall 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.

In the following description of redirections, references are made to opening and creating files. These references shall conform to the requirements in 2.9.1.4. A failure to open or create a file shall cause the redirection to fail.

3.7.1 Redirecting Input

Input redirection shall 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 shall refer to standard input (file descriptor `0`).

3.7.2 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 shall refer to standard output (file descriptor `1`).

Output redirection using the > format shall fail if the *noclobber* option is set (see 1
the description of `set -C` in 3.14.11) and the file named by the expansion of *word* 1
exists and is a regular file. Otherwise, redirection using the > or >| formats shall 1
cause the file whose name results from the expansion of *word* to be created and opened for ouput on the designated file descriptor, or standard output if none is specified. If the file does not exist, it shall be created; otherwise, it shall be truncated to be an empty file after being opened.

3.7.3 Appending Redirected Output

Appended output redirection shall 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 POSIX.1 {8} *open*() function was called with the O_APPEND flag. If the file does not exist, it shall be created.

The general format for appending redirected output is as follows:

> [*n*]>>*word*

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

3.7.4 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 shall be treated as a single word that begins after the next `<newline>` and continues until there is a line containing only the delimiter, with no trailing `<blank>`s. 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 shall be formed by performing quote removal on *word*, and the here-document lines shall not be expanded. Otherwise, the delimiter shall be the *word* itself.

If no characters in *word* are quoted, all lines of the here-document shall be expanded for parameter expansion, command substitution, and arithmetic expansion. In this case, the backslash in the input shall behave as the backslash inside double-quotes (see 3.2.3). However, the double-quote character (") shall not be treated specially within a here-document, except when the double-quote appears within `$( )`, `‘ ‘`, or `${ }`. 1

If the redirection symbol is `<<−`, all leading `<tab>` characters shall 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 shall be supplied first by the application and shall be read first by the shell.

3.7.5 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, shall 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 shall result (see 3.8.1). If *word* evaluates to −, file descriptor *n*, or standard input if *n* is not specified, shall be closed. If *word* evaluates to something else, the behavior is unspecified. 1

3.7.6 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, shall 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 shall result (see 3.8.1). If *word* evaluates to −, 1
file descriptor *n*, or standard output if *n* is not specified, shall be closed. If *word* evaluates to something else, the behavior is unspecified.

3.7.7 Open File Descriptors for Reading and Writing.

The redirection operator

[*n*]<>*word*

shall 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 shall be created.

3.7.8 Redirection Rationale. *(This subclause is not a part of P1003.2)*

In the C binding for POSIX.1 {8}, file descriptors are integers in the range 0 – ({OPEN_MAX}−1). The file descriptors discussed in Redirection are that same set of small integers.

As POSIX.2 is being finalized, it is not known how file descriptors will be represented in the language-independent description of POSIX.1 {8}. The current consensus appears to be that they will remain as small integers, but it is still possible that they will be defined as an opaque type. If they remain as integers, then the current POSIX.2 wording is acceptable. If they become an opaque type, then the C binding to POSIX.1 {8} will have to define the mapping from the binding's small integers to the opaque type, and the Redirection clause in POSIX.2 will have to be modified to specify that same mapping.

Having multidigit file descriptor numbers for I/O redirection can cause some obscure compatibility problems. Specifically, scripts that depend on an example command:

```
echo 22>/dev/null
```

echoing `"2"` are somewhat broken to begin with. However, the file descriptor number still must be delimited from the preceding text. For example,

```
cat file2>foo
```

will write the contents of `file2`, not the contents of `file`.

The `>|` format of output redirection was adopted from the KornShell. Along with the *noclobber* option, `set −C`, it provides a safety feature to prevent inadvertent overwriting of existing files. (See the rationale with the `pathchk` utility for why this step was taken.) The restriction on regular files is historical practice.

The System V shell and the KornShell have differed historically on pathname expansion of *word*; the former never performed it, the latter only when the result was a single field (file). As a compromise, it was decided that the KornShell functionality was useful, but only as a shorthand device for interactive users. No reasonable shell script would be written with a command such as:

```
cat foo > a*
```

Thus, shell scripts are prohibited from doing it, while interactive users can select the shell with which they are most comfortable.

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`.

The `<>` operator is a feature first documented in the KornShell, but it has been silently present in both System V and BSD shells. It 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.

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.

Tilde expansion is not performed on a here-document because the data is treated 1
as if it were enclosed in double-quotes. 1

3.8 Exit Status and Errors

3.8.1 Consequences of Shell Errors

For a noninteractive shell, an error condition encountered by a special built-in (see 3.14) or other type of utility shall cause the shell to write a diagnostic message to standard error and exit as shown in the following table:

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

An ‘‘expansion error’’ is one that occurs when the shell expansions defined in 3.6 are carried out (e.g., `${x!y}`, because `!` is not a valid operator); an implementation may treat these as syntax errors if it is able to detect them during tokenization, rather than during expansion.

If any of the errors shown as ‘‘shall (may) exit’’ occur in a subshell, the subshell shall (may) exit with a nonzero status, but the script containing the subshell shall not exit because of the error.

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

3.8.2 Exit Status for Commands

Each command has an exit status that can influence the behavior of other shell commands. The exit status of commands that are not utilities are documented in this subclause. The exit status of the standard utilities are documented in their respective clauses.

If a command is not found by the shell, the exit status shall be 127. If the command name is found, but it is not an executable utility, the exit status shall be 126. See 3.9.1.1. 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 shall be greater than zero.

Internally, for purposes of deciding if a command exits with a nonzero exit status, the shell shall recognize the entire status value retrieved for the command by the equivalent of the POSIX.1 {8} *wait*() function WEXITSTATUS macro. When reporting the exit status with the special parameter ?, the shell shall report the full eight bits of exit status available. The exit status of a command that terminated because it received a signal shall be reported as greater than 128.

3.8.3 Exit Status and Errors Rationale. *(This subclause is not a part of P1003.2)*

There is a historical difference in `sh` and `ksh` noninteractive error behavior. When a command named in a script is not found, some implementations of `sh` exit immediately, but `ksh` continues with the next command. Thus, POSIX.2 says that the shell ''may'' exit in this case. This puts a small burden on the programmer, who will have to test for successful completion following a command if it is important that the next command not be executed if the previous was not found. If it is important for the command to have been found, it was probably also important for it to complete successfully. The test for successful completion would not need to change.

Historically, shells have returned an exit status of 128+*n*, where *n* represents the signal number. Since signal numbers are not standardized, there is no portable way to determine which signal caused the termination. Also, it is possible for a command to exit with a status in the same range of numbers that the shell would use to report that the command was terminated by a signal. Implementations are encouraged to chose exit values greater than 256 to indicate programs that terminated by a signal so that the exit status cannot be confused with an exit status generated by a normal termination.

Historical shells make the distinction between ''utility not found'' and ''utility found but cannot execute'' in their error messages. By specifying two seldomly used exit status values for these cases, 127 and 126 respectively, this gives an application the opportunity to make use of this distinction without having to parse an error message that would probably change from locale to locale. The POSIX.2 `command`, `env`, `nohup`, and `xargs` utilities also have been specified to use this convention.

When a command fails during word expansion or redirection, most historical implementations exit with a status of 1. However, there was some sentiment that this value should probably be much higher, so that an application could distinguish this case from the more normal exit status values. Thus, the language ''greater than zero'' was selected to allow either method to be implemented.

3.9 Shell Commands

This clause 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 recognizing the command type, and does not formally represent the syntax. Each description discusses the semantics of the command; for a formal description of the command language, consult the grammar in 3.10.

A *command* is one of the following:

— *simple command* (see 3.9.1)

— *pipeline* (see 3.9.2)

— *list* or *compound-list* (see 3.9.3)

— *compound command* (see 3.9.4)

— *function definition* (see 3.9.5).

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}, etc.).

3.9.0.1 Shell Commands Rationale. *(This subclause is not a part of P1003.2)*

A description of an ‘‘empty command’’ was removed from an earlier draft because it is only relevant in the cases of `sh -c ""`, `system("")`, or an empty shell-script file (such as the implementation of `true` on some historical systems). Since it is no longer mentioned in POSIX.2, it falls into the silently unspecified category of behavior where implementations can continue to operate as they have historically, but conforming applications will not construct empty commands. (However, note that `sh` does explicitly state an exit status for an empty string or file.) In an interactive session or a script with other commands, extra `<newline>`s or semicolons, such as

```
$ false
$
$ echo $?
1
```

would not qualify as the empty command described here because they would be consumed by other parts of the grammar.

3.9.1 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 (i.e., 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 shall 1
all be performed from the beginning of the command text to the end.

(1) The words that are recognized as variable assignments or redirections according to 3.10.2 are saved for processing in steps (3) and (4).

(2) The words that are not variable assignments or redirections shall be expanded. If any fields remain following their expansion, the first field shall be considered the command name, and remaining fields shall be the arguments for the command.

(3) Redirections shall be performed as described in 3.7.

(4) Each variable assignment shall 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 3.14.

If no command name results, variable assignments shall affect the current execution environment. Otherwise, the variable assignments shall be exported for the execution environment of the command and shall 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 shall occur. See 3.8.1 for the consequences of these errors.

If there is no command name, any redirections shall 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 `exec` (3.14.6)]. If any of the redirections performed in the current shell execution environment fail, the command shall immediately fail with an exit status greater than zero, and the shell shall write an error message indicating the failure. See 3.8.1 for the consequences of these failures on interactive and noninteractive shells.

If there is a command name, execution shall continue as described in 3.9.1.1. If there is no command name, but the command contained a command substitution, the command shall complete with the exit status of the last command substitution performed. Otherwise, the command shall complete with a zero exit status.

3.9.1.0.1 Simple Commands Rationale. *(This subclause is not a part of P1003.2)*

The enumerated list is used only when the command is actually going to be exe- 1
cuted. For example, in: 1

```
true || $foo *
```
1

no expansions are performed. 1

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
```

Historical practice is for a command without a command name, but that includes a command substitution, to have an exit status of the last command substitution that the shell performed and some historical scripts rely on this. 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     1
        exit 1                                      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 the same ones. For example, in: 1

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

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

3.9.1.1 Command Search and Execution

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

(1) If the command name does not contain any slashes, the first successful step in the following sequence shall occur:

(a) If the command name matches the name of a special built-in utility, that special built-in utility shall be invoked.

(b) If the command name matches the name of a function known to this shell, the function shall be invoked as described in 3.9.5. [If the implementation has provided a standard utility in the form of a function, it shall not be recognized at this point. It shall be invoked in conjunction with the path search in step (1)(d).]

(c) If the command name matches the name of a utility listed in Table 2-2 (see 2.3), that utility shall be invoked.

(d) Otherwise, the command shall be searched for using the **PATH** environment variable as described in 2.6:

[1] If the search is successful:

[a] If the system has implemented the utility as a regular built-in or as a shell function, it shall be invoked at this point in the path search.

[b] Otherwise, the shell shall execute the utility in a separate utility environment (see 3.12) with actions equivalent to calling the POSIX.1 {8} *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 POSIX.1 {8} error [ENOEXEC], the shell shall 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 shall repeat the search to find the new location for the utility, if any.

[2] If the search is unsuccessful, the command shall fail with an exit status of 127 and the shell shall write an error message.

(2) If the command name does contain slashes, the shell shall execute the utility in a separate utility environment with actions equivalent to calling the POSIX.1 {8} *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 POSIX.1 {8} error [ENOEXEC], the shell shall 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.

3.9.1.1.1 Command Search and Execution Rationale. *(This subclause is not a part of P1003.2)*

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 command search shown here does not match all historical implementations. A more typical sequence has been:

— Any built-in, special or regular.

— Functions.

— Path search for executable files.

But there are problems with this sequence. Since the programmer has no idea in advance which utilities might have been built into the shell, a function cannot be used to portably override a utility of the same name. (For example, a function named `cd` cannot be written for many historical systems.) Furthermore, the **PATH** variable is partially ineffective in this case and only a pathname with a slash can be used to ensure a specific executable file is invoked.

The sequence selected for POSIX.2 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; see 4.12) 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 POSIX.2, 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 must ensure that all effects specified by POSIX.2 resulting from the invocation of the regular built-in or function (interaction with the environment, variables, traps, etc.) 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 her own utility named `ls` and puts it in `/hsa/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

After the *execve*() failure described, the shell normally executes the file as a shell script. Some implementations, however, attempt to detect whether the file is actually a script and not an executable from some other architecture. The method used by the KornShell is allowed by the text that indicates nontext files may be bypassed.

3.9.2 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 shall be connected to the standard input of the next command.

The format for a pipeline is:

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

The standard output of *command1* shall be connected to the standard input of *command2*. The standard input, standard output, or both of a command shall be considered to be assigned by the pipeline before any redirection specified by redirection operators that are part of the command (see 3.7).

If the pipeline is not in the background (see 3.9.3.1), the shell shall 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 shall 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 shall be 1; if the last command returns greater than zero, the exit status is zero.

3.9.2.1 Pipelines Rationale. *(This subclause is not a part of P1003.2)*

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 ! was added to allow more flexible testing using AND and OR lists.

It was suggested that it would be better to return a nonzero value if any command in the pipeline terminates with nonzero status (perhaps the bitwise OR of all return values). However, the choice of the last-specified command semantics are historical practice and would cause application breakage if changed. An example of historical (and POSIX. 2) behavior:

```
$ sleep 5 | (exit 4)
$ echo $?
4
$ (exit 4) | sleep 5        1
$ echo $?                   1
0                           1
```

3.9.3 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 `||` shall have equal precedence and shall be evaluated from beginning to end.

A `;` or `<newline>` terminator shall cause the preceding AND-OR-list to be executed sequentially; an & shall cause asynchronous execution of the preceding AND-OR-list.

The term *compound-list* is derived from the grammar in 3.10; it is equivalent to a sequence of *lists,* separated by `<newline>`s, that can be preceded or followed by an arbitrary number of `<newline>`s.

3.9.3.0.1 Lists Rationale. *(This subclause is not a part of P1003.2)*

The equal precedence of && and || is historical practice. The developers of the standard evaluated the model used more frequently in high level programming languages, such as C, to allow the shell logical operators to be used for complex expressions in an unambiguous way, but could not in the end allow existing scripts to break in the subtle way unequal precedence might cause. Some arguments were posed concerning the { } or ( ) groupings that are required historically. There are some disadvantages to these groupings:

— The ( ) can be expensive, as they spawn other processes on some systems. This performance concern is primarily an implementation issue.

— The { } braces are not operators (they are reserved words) and require a trailing space after each {, and a semicolon before each }. Most programmers (and certainly interactive users) have avoided braces as grouping constructs because of the irritating syntax required. Braces were not changed to operators because that would generate compatibility issues even greater than the precedence question; braces appear outside the context of a keyword in many shell scripts.

An example reiterates the precedence of the lists as they associate from beginning to end. Both of the following commands write solely `bar` to standard output:

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

The following is an example that illustrates `<newline>`s 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
```

3.9.3.1 Asynchronous Lists

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

The format for running a command in background is:

command1 & [*command2* & ...]

The standard input for an asynchronous list, before any explicit redirections are performed, shall 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 shall override this activity.

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 shall become known in the current shell execution environment; see 3.12. This process ID shall remain known until:

— The command terminates and the application waits for the process ID, or

— Another asynchronous list is 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 shall be zero.

3.9.3.1.1 Asynchronous Lists Rationale. *(This subclause is not a part of P1003.2)*

The grammar treats a construct such as

```
foo & bar & bam &
```

as one ‘‘asynchronous list,’’ but since the status of each element is tracked by the shell, the term ‘‘element of an asynchronous list’’ was introduced to identify just one of the `foo`, `bar`, `bam` portions of the overall list.

Unless the implementation has an internal limit, such as {CHILD_MAX}, on the retained process IDs, it would require unbounded memory for the following example:

```
while true
do      foo & echo $!
done
```

The treatment of the signals SIGINT and SIGQUIT with asynchronous lists is described in 3.11.

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
```

3.9.3.2 Sequential Lists

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

The format for executing commands sequentially is:

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

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

Exit Status

The exit status of a sequential list shall be the exit status of the last command in the list.

3.9.3.3 AND Lists

The control operator `&&` shall denote an AND list. The format is:

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

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

Exit Status

The exit status of an AND list shall be the exit status of the last command that is executed in the list.

3.9.3.4 OR Lists

The control operator `||` shall denote an OR List. The format is:

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

First, *command1* is executed. If its exit status is nonzero, *command2* is 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 shall be the exit status of the last command that is executed in the list.

3.9.4 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 shall apply to all the commands within the compound command that do not explicitly override that redirection.

3.9.4.1 Grouping Commands

The format for grouping commands is as follows:

`(`*compound-list*`)`	Execute *compound-list* in a subshell environment; see 3.12. Variable assignments and built-in commands that affect the environment shall not remain in effect after the list finishes.
`{` *compound-list* `;}`	Execute *compound-list* in the current process environment.

Exit Status

The exit status of a grouping command shall be the exit status of *list*.

3.9.4.1.1 Grouping Commands Rationale. *(This subclause is not a part of P1003.2)*

The semicolon shown in `{` *compound-list* `;}` is an example of a control operator delimiting the `}` reserved word. Other delimiters are possible, as shown in 3.10; `<newline>` is frequently used.

A proposal was made to use the <do-done> construct in all cases where command grouping performed in the current process environment is performed, identifying it as a construct for the grouping commands, as well as for shell functions. This was not included because the shell already has a grouping construct for this purpose (`{ }`), and changing it would have been counter-productive.

3.9.4.2 `for` Loop

The `for` loop shall 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` shall be expanded to generate a list of items. Then, the variable *name* shall be set to each item, in turn, and the *compound-list* executed each time. If no items result from the expansion, the *compound-list* shall not be executed. Omitting

```
in word ...
```

is equivalent to

```
in "$@"
```

Exit Status

The exit status of a `for` command shall be the exit status of the last command that executes. If there are no items, the exit status shall be zero.

3.9.4.2.1 `for` Loop Rationale. *(This subclause is not a part of P1003.2)*

The format is shown with generous usage of `<newline>`s. See the grammar in 3.10 for a precise description of where `<newline>`s and semicolons can be interchanged.

Some historical implementations support `{` and `}` as substitutes for `do` and `done`.
The working group chose to omit them, even as an obsolescent feature. (Note that
these substitutes were only for the `for` command; the `while` and `until` com- 1
mands could not use them historically, because they are followed by compound- 1
lists that may contain `{...}` grouping commands themselves 1

The reserved word pair `do` ... `done` was selected rather than `do` ... `od` (which would have matched the spirit of `if` ... `fi` and `case` ... `esac`) because `od` is a commonly-used utility name and this would have been an unacceptable choice.

3.9.4.3 `case` Conditional Construct

The conditional construct `case` shall execute the *compound-list* corresponding to the first one of several *patterns* (see 3.13) 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` shall 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 ;;          2
     [(]pattern2 | pattern3 ) compound-list ;;          2
     ...
esac
```

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

Each pattern in a pattern list shall be expanded and compared against the expansion of *word*. After the first match, no more patterns shall be expanded, and the *compound-list* shall be executed. The order of expansion and comparing of patterns in a multiple pattern list is unspecified.

Exit Status

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

3.9.4.3.1 `case` Conditional Construct Rationale. *(This subclause is not a part of P1003.2)*

An optional open-parenthesis before *pattern* was added to allow numerous historical KornShell scripts to conform. At one time, using the leading parenthesis was required if the `case` statement were to be embedded within a `$( )` command substitution; this is no longer the case with the POSIX shell. Nevertheless, many existing scripts use the open-parenthesis, if only because it makes matching-parenthesis searching easier in `vi` and other editors. This is a relatively simple implementation change that is fully upward compatible for all scripts.

Consideration was given to requiring `break` inside the *compound-list* to prevent falling through to the next pattern action list. This was rejected as being nonexisting practice. An interesting undocumented feature of the KornShell is that using `;&` instead of `;;` as a terminator causes the exact opposite behavior—the flow of control continues with the next *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.

3.9.4.4 `if` Conditional Construct

The `if` command shall 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 shall 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 shall complete. Otherwise, the `else` *compound-list* is executed.

Exit Status

The exit status of the `if` command shall be the exit status of the `then` or `else` *compound-list* that was executed, or zero, if none was executed.

3.9.4.4.1 `if` Conditional Construct Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.4.5 `while` Loop

The `while` loop continuously shall 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* shall be executed, and if it has a nonzero exit status, the `while` command shall complete. Otherwise, the *compound-list-2* shall be executed, and the process shall repeat.

Exit Status

The exit status of the `while` loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

3.9.4.5.1 `while` Loop Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.4.6 `until` Loop

The `until` loop continuously shall execute one *compound-list* as long as another *compound-list* has a nonzero exit status.

The format of the `until` loop is as follows

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

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

Exit Status

The exit status of the `until` loop shall be the exit status of the last *compound-list-2* executed, or zero if none was executed.

3.9.4.6.1 `until` Loop Rationale. *(This subclause is not a part of P1003.2)*

The precise format for the command syntax is described in 3.10.

3.9.5 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 shall be a name (see 3.1.5). An implementation 1
may allow other characters in a function name as an extension. The implementa- 1
tion shall maintain separate namespaces for functions and variables.

The argument *compound-command* represents a compound command, as described in 3.9.4.

When the function is declared, none of the expansions in 3.6 shall be performed on the text in *compound-command* or *io-redirect* ; all expansions shall be performed as normal each time the function is called. Similarly, the optional *io-redirect* redirections and any variable assignments within *compound-command* shall be performed during the execution of the function itself, not the function definition. See 3.8.1 for the consequences of failures of these operations on interactive and noninteractive shells.

When a function is executed, it shall have the syntax-error and variable-assignment properties described for special built-in utilities, in the enumerated list at the beginning of 3.14.

The *compound-command* shall be executed whenever the function name is specified as the name of a simple command (see 3.9.1.1). The operands to the command temporarily shall become the positional parameters during the execution of the *compound-command* ; the special parameter `#` shall also be changed to reflect the number of operands. The special parameter `0` shall be unchanged. When the function completes, the values of the positional parameters and the special parameter `#` shall 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 shall complete and execution shall resume with the next command after the function call.

Exit Status

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

3.9.5.1 Function Definition Command Rationale *(This subclause is not a part of P1003.2)*

The description of functions in Draft 8 was based on the notion that functions should behave like miniature shell scripts; that is, except for sharing variables, most elements of an execution environment should behave as if it were a new execution environment, and changes to these should be local to the function. For example, traps and options should be reset on entry to the function, and any changes to them don't affect the traps or options of the caller. There were numerous objections to this basic idea, and the opponents asserted that functions were intended to be a convenient mechanism for grouping commonly executed commands that were to be executed in the current execution environment, similar to the execution of the `dot` special built-in.

Opponents also pointed out that the functions described in Draft 8 did not scope everything a new shell script would anyway, such as the current working directory, or umask, but instead picked a few select properties. The basic argument was that if one wanted scoping of the execution environment, the mechanism already exists: put the commands in a new shell script and call it. All traditional shells that implemented functions, other than the KornShell, have implemented functions that operate in the current execution environment. Because of this, Draft 9 removed any local scoping of traps or options. Local variables within a function were considered and included in Draft 9 (controlled by the special built-in `local`), but were removed because they do not fit the simple model developed for the scoping of functions and there was some opposition to adding yet another new special built-in from outside existing practice. Implementations should reserve the identifier `local` (as well as `typeset`, as used in the KornShell) in case this local variable mechanism is adopted in a future version of POSIX.2.

A separate issue from the execution environment of a function is the availability of that function to child shells. A few objectors, including the author of the original Version 7 UNIX system shell, maintained that just as a variable can be shared with child shells by exporting it, so should a function—and so this capability has been added to the standard. In previous drafts, the `export` command therefore had a `-f` flag for exporting functions. Functions that were exported were to be put into the environment as *name* `()=`*value* pairs, and upon invocation, the shell would scan the environment for these, and automatically define these functions. This facility received a lot of balloting opposition and was removed from Draft 11. Some of the arguments against exportable functions were:

— There was little existing practice. The Ninth Edition shell provided them, but there was controversy over how well it worked.

— There are numerous security problems associated with functions appearing in a script's environment and overriding standard utilities or the application's own utilities.

— There was controversy over requiring `make` to import functions, where it has historically used an *exec* function for many of its command line executions.

— Functions can be big and the environment is of a limited size. (The counter-argument was that functions are no different than variables in terms of size: there can be big ones, and there can be small ones—and just as one does not export huge variables, one does not export huge functions. However, this insight might be lost on the average shell-function writer, who typically writes much larger functions than variables.)

As far as can be determined, the functions in POSIX.2 match those in System V. The KornShell has two methods of defining functions:

`function` *fname* `{` *compound-list* `}`

and

fname `() {` *compound-list* `}`

The latter uses the same definition as POSIX.2, but differs in semantics, as described previously. A future edition of the KornShell is planned to align the latter syntax with POSIX and keep the former as-is.

The name space for functions is limited to that of a *name* because of historical 1
practice. Complications in defining the syntactic rules for the function definition 1
command and in dealing with known extensions such as the KornShell's `@()` 1
prevented the name space from being widened to a *word*, as requested by some 1
balloters. Using functions to support synonyms such as the C-shell's `!!` and `%` is 1
thus disallowed to portable applications, but acceptable as an extension. For 1
interactive users, the aliasing facilities in the UPE should be adequate for this 1
purpose. It is recognized that the name space for utilities in the file system is 1
wider than that currently supported for functions, if the portable filename charac- 1
ter set guidelines are ignored, but it did not seem useful to mandate extensions in 1
systems for so little benefit to portable applications. 1

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

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
#
[ X$i = Xyes ] && foo() {
        ls -l
}
```

3.10 Shell Grammar

The following grammar describes the Shell Command Language. Any discrepancies found between this grammar and the preceding description shall be resolved in favor of this clause.

3.10.1 Shell Grammar Lexical Conventions

The input language to the shell must be first recognized at the character level. The resulting tokens shall 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 3.3 shall apply.

(1) A `<newline>` shall be returned as the token identifier `NEWLINE`.

(2) If the token is an operator, the token identifier for that operator shall result.

(3) If the string consists solely of digits and the delimiter character is one of < or >, the token identifier `IO_NUMBER` shall be returned.

(4) Otherwise, the token identifier `TOKEN` shall 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 shall be applied to convert the token identifier type of the `TOKEN` to a token identifier acceptable at that point in the grammar. The reduction shall then proceed based upon the token identifier type yielded by the rule applied. When more than one rule applies, the highest numbered rule shall 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 shall have the word expansion rules applied to them immediately before the associated command is executed, not at the time the command is parsed.

3.10.2 Shell Grammar Rules

(1) [Command Name]
When the `TOKEN` is exactly a reserved word, the token identifier for that
reserved word shall result. Otherwise, the token `WORD` shall be returned. 1
Also, if the parser is in any state where only a reserved word could be the 1
next correct token, proceed as above. 1

NOTE: Because at this point quote marks are retained in the token, quoted strings cannot be recognized as reserved words. This rule also implies that reserved words will not be recognized except in certain positions in the input, such as after a `<newline>` 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 tokenization, as described in 3.2.1, so escaped newlines are already removed at this point.

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

(2) [Redirection to/from filename]
The expansions specified in 3.7 shall occur. As specified there, exactly
one field can result (or the result is unspecified), and there are additional 1
requirements on pathname expansion.

(3) [Redirection from here-document]
Quote removal [3.7.4]. shall be applied to the word to determine the del- 1
imiter that will be used to find the end of the here-document that begins 1
after the next `<newline>`. 1

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

(5) [`NAME` in `for`]
When the `TOKEN` meets the requirements for a name [3.1.5], the token identifier `NAME` shall result. Otherwise, the token `WORD` shall be returned.

(6) [Third word of `for` and `case`]
When the `TOKEN` is exactly the reserved word `In`, the token identifier for `In` shall result. Otherwise, the token `WORD` shall be returned.

(7) [Assignment preceding command name] 1

(a) [When the first word]
If the `TOKEN` does not contain the character `=`, rule (1) shall be applied. Otherwise, apply (7)(b).

(b) [Not the first word]
If the `TOKEN` contains the equals-sign character:

— If it begins with `=`, the token `WORD` shall be returned.

— If all the characters preceding `=` form a valid name [3.1.5], the token `ASSIGNMENT_WORD` shall 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` shall occur as specified in 3.9.1.

(8) [`NAME` in function]
When the `TOKEN` is exactly a reserved word, the token identifier for that reserved word shall result. Otherwise, when the `TOKEN` meets the requirements for a name [3.1.5], the token identifier `NAME` shall result. Otherwise, rule (7) shall apply.

(9) [Body of function]
Word expansion and assignment shall 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 shall instead be returned as a single `WORD` consisting only of characters that are exactly the token described in 3.3.

```
/*      -------------------------------------------------------
        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
        recognized when reserved words are recognized.          */

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

%token  In
/*      'in'                                                    */

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

%start  complete_command

%%

complete_command    : list separator
                    | list                                          1
                    ;

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                            do_group
                    | For name In wordlist sequential_sep do_group
                    ;
name                : NAME                   /* Apply rule (5) */           2
                    ;
in                  : In                     /* Apply rule (6) */
                    ;
wordlist            : wordlist WORD
                    |          WORD
                    ;
case_clause         : Case WORD In linebreak case_list Esac
                    | Case WORD 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          2
                    | '(' pattern ')' compound_list DSEMI linebreak          2
                    ;
pattern             :            WORD          /* Apply rule (4) */
                    | pattern '|' WORD         /* Do not apply rule (4) */   1
                    ;
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) */   2
                    ;
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 (7)(a) */
                    ;
cmd_word            : WORD                  /* Apply rule (7)(b) */
                    ;
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
                     ;
```

3.10.3 Shell Grammar Rationale. *(This subclause is not a part of P1003.2)*

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 recognized, 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).

Concerning the third sentence of rule (1) (''Also, if the parser ...''):

— This sentence 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 rule (1)] would apply. In many instances the two conditions are identical, but this part of rule (1) does not give license to treating a `WORD` as a reserved words unless it is in a place where a reserved word must appear.

— The statement is equivalent to requiring that when the LR(1) lookahead set contains exactly a reserved word, it must be recognized if it is present. (Here 'LR(1)'' refers to the theoretical concepts, not to any real parser generator.)

For example, in the construct below, and when the parser is at the point marked with ˆ, the only next legal token is `then` (this follows directly from the grammar rules).

```
if if....fi then ....  fi
           ˆ
```

At that point, the `then` must be recognized as a reserved word.

(Depending on the actual parser generator actually used, ''extra'' reserved words may be in some lookahead sets. It does not really matter if they are recognized, or even if any possible reserved word is recognized in that state, because if it is recognized and is not in the (theoretical) LR(1) lookahead set, an error will ultimately be detected. In the example above, if some other reserved word (e.g., `while`) is also recognized, an error will occur later.

This is approximately equivalent to saying that reserved words are recognized after other reserved words (because it is after a reserved word that this condition will occur), but avoids the ''except for...'' list that would be required for `case`, `for`, etc. (Reserved words are of course recognized anywhere a `simple_command` can appear, as well. Other rules take care of the special cases of nonrecognition, such as rule (4) for case statements.)

Note that the body of here-documents are handled by Token Recognition (see 3.3) 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.

The processing of here-documents is handled as part of token recognition (see 3.3) rather than as part of the grammar.

3.11 Signals and Error Handling

When a command is in an asynchronous list, the shell shall prevent SIGQUIT and SIGINT signals from the keyboard from interrupting the command. Otherwise, signals shall have the values inherited by the shell from its parent (see also 3.14.13).

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 shall 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 shall cause the `wait` utility to return immediately with an exit status >128, immediately after which the trap associated with that signal shall be taken.

If multiple signals are pending for the shell for which there are associated trap actions (see 3.14.13), the order of execution of trap actions is unspecified.

3.12 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` (see 4.5).
- File Creation Mask set by `umask` (see 4.67).
- Current traps set by `trap` (see 3.14.13).
- Shell parameters that are set by variable assignment (see `set` in 3.14.11) or from the POSIX.1 {8} environment inherited by the shell when it begins (see `export` in 3.14.8).
- Shell functions (see 3.9.5.)
- Options turned on at invocation or by `set`.
- Process IDs of the last commands in asynchronous lists known to this shell environment; see 3.9.3.1.

Utilities other than the special built-ins (see 3.14) shall be invoked in a separate environment that consists of the following. The initial value of these objects shall 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 (see 3.14.6), 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 shall be set to the default values and traps ignored by the shell shall be set to be ignored by the utility. If the utility is not a shell script, the trap actions (default or ignore) shall 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, shall be passed to the utility as POSIX.1 {8} environment variables.

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

A subshell environment shall be created as a duplicate of the shell environment, 1
except that signal traps set by that shell environment shall be set to the default 1
values. Changes made to the subshell environment shall not affect the shell environment. Command substitution, commands that are grouped with parentheses, and asynchronous lists shall be executed in a subshell environment. Additionally, each command of a multicommand 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 shall be executed in the current shell environment.

3.12.0.1 Shell Execution Environment Rationale. *(This subclause is not a part of P1003.2)*

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. It was decided to allow this extension, but not require it; therefore, a shell programmer should consider a pipeline to be in a subshell environment, but not depend on it.

The previous description of execution environment failed to mention that each command in a multiple command pipeline could be in a subshell execution environment. For compatibility with some existing shells, the wording was phrased to allow an implementation to place any or all commands of a pipeline in the current environment. However, this means that a POSIX application must assume each command is in a subshell environment, but not depend on it.

The wording about shell scripts is meant to convey the fact that describing ‘‘trap actions’’ can only be understood in the context of the shell command language. Outside this context, such as in a C-language program, signals are the operative condition, not traps.

3.13 Pattern Matching Notation

The pattern matching notation described in this clause 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 2.8. For this reason, the description of the rules for this pattern matching notation are based on the description of regular expression notation.

3.13.0.1 Pattern Matching Notation Rationale. *(This subclause is not a part of P1003.2)*

Pattern matching is a simpler concept and has a simpler syntax than regular expressions, as the former is generally used for the manipulation of file names, which are relatively simple collections of characters, while the latter is generally used to manipulate arbitrary text strings of potentially greater complexity. However, some of the basic concepts are the same, so this clause points liberally to the detailed descriptions in 2.8.

3.13.1 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 also shall match a single collating element.

An ordinary character is a pattern that shall match itself. It can be any character
in the supported character set except for NUL, those special shell characters in 3.2 1
that require quoting, and the following three special pattern characters. Match- 1
ing shall be based on the bit pattern used for encoding the character, not on the 1
graphic representation of the character. If any character (ordinary, shell special, 1
or pattern special) is quoted, that pattern shall match the character itself. The 1
shell special characters always require quoting. 1

When unquoted and outside a bracket expression, the following three characters 1
shall have special meaning in the specification of patterns: 1

- `?` A question-mark is a pattern that shall match any character.
- `*` An asterisk is a pattern that shall match multiple characters, as described in 3.13.2.
- `[` The open bracket shall introduce a pattern bracket expression.

The description of basic regular expression bracket expressions in 2.8.3.2 also shall apply to the pattern bracket expression, except that the exclamation-mark character (`!`) shall replace the circumflex character (`^`) in its role in a *nonmatching list* in the regular expression notation. A bracket expression starting with an unquoted circumflex character produces unspecified results.

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 an *exec* function, 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>`. This escaping `<backslash>` shall be discarded. The sequence `\\` shall represent one literal backslash. All of the requirements and effects of quoting on ordinary, shell special, and special pattern characters shall apply to escaping in this context.

3.13.1.1 Patterns Matching a Single Character Rationale. *(This subclause is not a part of P1003.2)*

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?c`, `a*c`. The following will not: `"a?c"`, `a\*c`, `a\[b]c`, `a["\b"]c`.

— Calling a utility or function without going through a shell, as described for `find` and *fnmatch* ().

— Calling utilities such as `find` or `pax` through the shell command line. (Although `find` and `pax` are the only instances of this in the standard utilities, describing it globally here is useful for future utilities that may use pattern matching internally.) 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 (called ‘‘metacharacters’’ in some historical documentation). 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.

The restriction on circumflex in a bracket expression is to allow implementations that support pattern matching using circumflex as the negation character in addition to the exclamation-mark. 1

3.13.2 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 shall match any string, including the null string.

(2) The concatenation of *patterns matching a single character* is a valid pattern that shall 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 shall 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.

3.13.2.1 Patterns Matching Multiple Characters Rationale. *(This subclause is not a part of P1003.2)*

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`.

3.13.3 Patterns Used for Filename Expansion

The rules described so far in 3.13.1 and 3.13.2 are qualified by the following rules that apply when pattern matching notation is used for filename expansion.

(1) The slash character in a pathname shall 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.

(2) If a filename begins with a period (`.`), the period shall be explicitly matched by using a period as the first character of the pattern or

immediately following a slash character. The leading period shall not be matched by:

— The asterisk or question-mark special characters, or

— A bracket expression containing a nonmatching 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 that does not contain a pattern character requires search permission. For example, given the pattern

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

search permission is needed for directory `/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 shall 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 shall be left unchanged.

3.13.3.1 Patterns Used for File Name Expansion Rationale. *(This subclause is not a part of P1003.2)*

The caveat about a slash within a bracket expression is derived from historical practice. 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`.

Filenames beginning with a period historically have been specially protected from view on UNIX systems. A proposal to allow an explicit period in a bracket expression to match a leading period was considered; it is allowed as an implementation extension, but a conforming application cannot make use of it. If this extension becomes popular in the future, it will be considered for a future version of POSIX. 2.

Historical systems have varied in their permissions requirements. To match `f*/bar` has required read permissions on the `f*` directories in the System V shell, but this standard, the C-shell, and KornShell require only search permissions.

3.14 Special Built-in Utilities

The following *special built-in* utilities shall be supported in the shell command language. The output of each command, if any, shall 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:

(1) 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 shall not cause a shell executing that utility to abort. (See 3.8.1 for the consequences of errors on interactive and noninteractive shells.) If a special built-in utility encountering a syntax error does not abort the shell, its exit value shall be nonzero.

(2) Variable assignments specified with special built-in utilities shall remain in effect after the built-in completes; this shall not be the case with a regular built-in or other utility.

As described in 2.3, the special built-in utilities in this clause need not be provided in a manner accessible via the POSIX.1 {8} *exec* family of functions.

Some of the special built-ins are described as conforming to the utility argument syntax guidelines in 2.10.2. For those that are not, the requirement in 2.11.3 that `"--"` be recognized as a first argument to be discarded does not apply and a conforming application shall not use that argument.

3.14.1 `break` — Exit from `for`, `while`, or `until` loop

`break` [*n*]

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 ≥ 1. The default is equivalent to $n=1$. If *n* is greater than the number of enclosing loops, the last enclosing loop shall be exited from. Execution continues with the command immediately following the loop.

Exit Status

0 Successful completion.

>0 The *n* value was not an unsigned decimal integer ≥ 1.

3.14.1.1 `break` Rationale. *(This subclause is not a part of P1003.2)*

Example:

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

Consideration was given to expanding the syntax of the `break` and `continue` to refer to a label associated with the appropriate loop, as a preferable alternative to the [*n*] method. This new method was proposed late in the development of the standard and adequate consensus could not be formed to include it. However, POSIX.2 does reserve the namespace of command names ending with a colon. It is anticipated that a future implementation could take advantage of this and provide something like:

```
outofloop: for i in a b c d e                                        1
do
        for j in 0 1 2 3 4 5 6 7 8 9
        do
                if test -r "${i}${j}"
                then break outofloop
                fi
        done
done
```

and that this might be standardized after implementation experience is achieved.

3.14.2 colon — Null utility

: [*argument* ...]

This utility shall only expand command *argument* s.

Exit Status

Zero.

3.14.2.1 colon Rationale. *(This subclause is not a part of P1003.2)*

The colon (`:`), or null utility, 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.

Example:

```
: ${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`.

3.14.3 `continue` — Continue `for`, `while`, or `until` loop

`continue` [*n*]

The `continue` utility shall 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 ≥ 1. The default is equivalent to *n*=`1`. If *n* is greater than the number of enclosing loops, the last enclosing loop is used.

Exit Status

0 Successful completion. 2

>0 The *n* value was not an unsigned decimal integer ≥ 1. 2

3.14.3.1 `continue` Rationale. *(This subclause is not a part of P1003.2)*

Example:

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

3.14.4 `dot` — Execute commands in current environment

`.` *file*

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

If *file* does not contain a slash, the shell shall 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 noninteractive shell shall abort; an interactive shell shall write a diagnostic message to standard error, but this condition shall not be considered a syntax error.

Exit Status

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

3.14.4.1 dot Rationale. *(This subclause is not a part of P1003.2)*

Some older implementations searched the current directory for the *file*, even if the value of **PATH** disallowed it. This behavior was omitted from POSIX.2 due to concerns about introducing the susceptibility to trojan horses that the user might be trying to avoid by leaving dot out of **PATH**.

The KornShell version of `dot` takes optional arguments that are set to the posi- 1
tional parameters. This is a valid extension that allows a dot script to behave 1
identically to a function.

Example:

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

3.14.5 `eval` — Construct command by concatenating arguments

`eval` [*argument* ...]

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

Exit Status

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

3.14.5.1 `eval` Rationale. *(This subclause is not a part of P1003.2)*

Example :

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

3.14.6 `exec` — Execute commands and open, close, and/or copy file descriptors

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

The `exec` utility opens, closes, and/or copies 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.

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

Exit Status

If *command* is specified, `exec` shall not return to the shell; rather, the exit status 2
of the process shall be the exit status of the program implementing *command* , 2
which overlaid the shell. If *command* is not found, the exit status shall be 127. If 1
command is found, but it is not an executable utility, the exit status shall be 126. 1
If a redirection error occurs (see 3.8.1), the shell shall exit with a value in the 1
range 1–125. Otherwise, `exec` shall return a zero exit status.

3.14.6.1 `exec` Rationale. *(This subclause is not a part of P1003.2)*

Most historical implementations are not conformant in that

```
foo=bar exec cmd
```

does not pass `foo` to `cmd`.

Earlier drafts stated that ‘‘If specified without *command* or *argument* , the shell sets to close-on-exec file numbers greater than 2 that are opened in this way, so that they will be closed when the shell invokes another program.’’ This was based on the behavior of one version of the KornShell and was made unspecified when it was realized that some existing scripts relied on the more generally historical behavior (leaving all file descriptors open). Furthermore, since the application should have no cognizance of whether a new shell is simply *fork* ()ed, rather than *exec*()ed, it could not consistently rely on the automatic closing behavior anyway. Scripts concerned that child shells could misuse open file descriptors can always close them explicitly, as shown in one of the following examples.

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
```

3.14.7 `exit` — Cause the shell to exit

exit [*n*]

The `exit` utility shall cause the shell to exit with the exit status specified by the 1
unsigned decimal integer *n*. If *n* is specified, but its value is not between 0 and 1
255 inclusively, the exit status is undefined. 1

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

Exit Status

The exit status shall be *n*, if specified. Otherwise, the value shall be the exit value of the last command executed, or zero if no command was executed. When `exit` is executed in a trap action (see 3.14.13), the ''last command'' is considered to be the command that executed immediately preceding the trap action.

3.14.7.1 `exit` Rationale. *(This subclause is not a part of P1003.2)*

As explained in other clauses, certain exit status values have been reserved for 1
special uses and should be used by applications only for those purposes: 1

126 A file to be executed was found, but it was not an executable utility. 1

127 A utility to be executed was not found. 1

>128 A command was interrupted by a signal. 1

Examples:

Exit with a *true* value:

```
exit 0
```

Exit with a *false* value:

```
exit 1
```

3.14.8 `export` — Set export attribute for variables

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

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

When `-p` is specified, `export` shall write to the standard output the names and values of all exported variables, in the following format: 1

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

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

The `export` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

Zero.

3.14.8.1 `export` Rationale. *(This subclause is not a part of P1003.2)*

When no arguments are given, the results are unspecified. Some historical shells use the no-argument case as the functional equivalent of what is required here with `-p`. This feature was left unspecified because it is not existing practice in all shells and some scripts may rely on the now-unspecified results on their implementations. Attempts to specify the `-p` output as the default case were unsuccessful in achieving consensus. The `-p` option was added to allow portable access to the values that can be saved and then later restored using, for instance, a dot script.

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
```

3.14.9 `readonly` — Set read-only attribute for variables 1

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

The variables whose *name* s are specified shall 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.

When `−p` is specified, `readonly` shall write to the standard output the names and values of all read-only variables, in the following format: 1

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

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

The `readonly` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

Zero.

3.14.9.1 `readonly` Rationale. *(This subclause is not a part of P1003.2)*

Example :

```
readonly HOME PWD
```

Some versions of the shell exist that preserve the read-only attribute across separate invocations. POSIX. 2 allows this behavior, but does not require it.

See the rationale for `export` (3.14.8.1) for a description of the no-argument and `−p` output cases.

In a previous draft, read-only functions were considered, but they were omitted as not being existing practice or particularly useful. Furthermore, functions must not be `readonly` across invocations to preclude *spoofing* (spoofing is the term for the practice of creating a program that acts like a well-known utility with the intent of subverting the user's real intent) of administrative or security-relevant (or -conscious) shell scripts.

3.14.10 `return` — Return from a function

```
return [n]
```

The return utility shall cause the shell to stop executing the current function or dot script (see 3.14.4). If the shell is not currently executing a function or dot script, the results are unspecified.

Exit Status

The value of the special parameter ? shall 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 (see 3.14.13), the ‘‘last command’’ is considered to be the command that executed immediately preceding the trap action.

3.14.10.1 `return` Rationale. *(This subclause is not a part of P1003.2)*

The behavior of `return` when not in a function or dot script differs between the System V shell and the KornShell. In the System V shell this is an error, whereas in the KornShell, the effect is the same as `exit`.

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 in 3.14.7.1.

3.14.11 `set` — Set/unset options and positional parameters

```
set [-aCefnuvx] [argument ...]
set [+aCefnuvx] [argument ...]
set –– [argument ...]
```

Obsolescent version :

```
set – [argument ...]
```

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

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

The *value* string shall be written with appropriate quoting so that it is suitable for re-input to the shell, (re)setting, as far as possible, the variables that are currently set. Readonly variables cannot be reset. See the description of shell quoting in 3.2.

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

The `set` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that options can be specified with either a leading hyphen (meaning enable the option) or plus-sign (meaning disable it).

The implementation shall 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`; see 4.56.

`−a` When this option is on, the export attribute shall be set for each variable to which an assignment is performed. (See 3.1.15.) If the assignment precedes a utility name in a command, the export attributes shall not persist in the current execution environment after the utility completes, with the exception that preceding one of the special built-in utilities shall 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 (see 4.27 and 4.52), the export attribute shall persist until the variable is unset.

`−C` (Uppercase C.) Prevent existing files from being overwritten by the shell's `>` redirection operator (see 3.7.2); the `>|` redirection operator shall 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 3.8.1 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 immediately shall exit.

`−f` The shell shall disable pathname expansion.

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

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

`−v` The shell shall write its input to standard error as it is read.

`−x` The shell shall 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` in 4.56). All the positional parameters shall be unset before any new values are assigned.

The remaining arguments shall be assigned in order to the positional parameters. The special parameter # shall 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* s shall unset all positional parameters and set the special parameter # to zero.

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

Exit Status

Zero.

3.14.11.1 `set` Rationale. *(This subclause is not a part of P1003.2)*

The `set −−` form is listed specifically in the Synopsis even though this usage is implied by the utility syntax guidelines. The explanation of this feature removes any ambiguity about whether the `set −−` form might be misinterpreted as being equivalent to `set` without any options or arguments. The functionality of this form has been adopted from the KornShell. In System V, `set −−` only unsets parameters if there is at least one argument; the only way to unset all parameters is to use `shift`. Using the KornShell version should not affect System V scripts because there should be no reason to deliberately issue it without arguments; if it were issued as, say:

```
set −− "$@"
```
1

and there were in fact no arguments resulting from `$@`, unsetting the parameters 1
would be a no-op anyway.

The `set +` form in earlier drafts was omitted as being an unnecessary duplication of `set` alone and not widespread historical practice.

The noclobber option was changed to `−C` from the `set -o noclobber` option in previous drafts. The `set −o` is used in the KornShell to accept word-length option names, duplicating many of the single-letter names. The noclobber option was changed to a single letter so that the historical `$-` paradigm would not be broken; see 3.5.2.

The following `set` flags were intentionally omitted with the following rationale:

`−h` This flag is related to command name hashing, which is not required for an implementation. It is primarily a performance issue, which is outside the scope of this standard.

−k The −k flag was originally added by Bourne to make it easier for users of prerelease versions of the shell. In early versions of the Bourne shell the construct set name=value, had to be used to assign values to shell variables. The problem with −k is that the behavior affects parsing, virtually precluding writing any compilers. To explain the behavior of −k, it is necessary to describe the parsing algorithm, which is implementation defined. For example,

```
set -k; echo name=value
```

and

```
set -k
echo name=value
```

behave differently. The interaction with functions is even more complex. What is more, the −k flag is never needed, since the command line could have been reordered.

−t The −t flag is hard to specify and almost never used. The only known use could be done with here-documents. Moreover, the behavior with ksh and sh differ. The man page says that it exits after reading and executing one command. What is one command? If the input is date;date, sh executes both date commands, ksh does only the first.

Consideration was given to rewriting set to simplify its confusing syntax. A specific suggestion was that the unset utility should be used to unset options instead of using the non-*getopt* ()-able +*option* syntax. However, the conclusion was reached that people were satisfied with the existing practice of using +*option* and there was no compelling reason to modify such widespread existing practice.

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
```

3.14.12 `shift` — Shift positional parameters

```
shift [n]
```

The positional parameters shall be shifted. Positional parameter 1 shall be assigned the value of parameter (1+*n*), parameter 2 shall be assigned the value of parameter (2+*n*), and so forth. The parameters represented by the numbers `$#` down to `$#`−*n*+1 shall be unset, and the parameter # shall be updated to reflect the new number of positional parameters.

The value *n* shall be an unsigned decimal integer less than or equal to the value of the special parameter #. If *n* is not given, it shall be assumed to be `1`. If *n* is `0`, the positional and special parameters shall not be changed.

Exit Status

The exit status shall be >0 if *n*>`$#`; otherwise, it shall be zero.

3.14.12.1 `shift` Rationale. *(This subclause is not a part of P1003.2)*

Example :

```
set a b c d e
shift 2
echo $*
c d e
```

3.14.13 `trap` — Trap signals

```
trap [action condition ...]
```

If *action* is −, the shell shall reset each *condition* to the default value. If *action* is null (`''`), the shell shall ignore each of the specified *condition* s if they arise. Otherwise, the argument *action* shall be read and executed by the shell when one of the corresponding conditions arises. The action of the trap shall override a previous action (either default action or one explicitly set). The value of `$?` after the trap action completes shall 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 Required Signals and Job Con- 1
trol Signals (Table 3-1 and Table 3-2 in POSIX.1 {8}). (For example: `HUP`, `INT`, `QUIT`, `TERM`). Setting a trap for SIGKILL or SIGSTOP produces undefined results.

The environment in which the shell executes a trap on `EXIT` shall 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 shall be processed in a manner equivalent to:

```
eval "$action"
```

Signals that were ignored on entry to a noninteractive 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 shall remain in place for a given shell until explicitly changed with another `trap` command.

The `trap` command with no arguments shall write to standard output a list of commands associated with each condition. The format is:

`"trap -- %s %s ...\n"`, *<action>*, *<condition>* ... 1

The shell shall format the output, including the proper use of quoting, so that it is suitable for re-input to the shell as commands that achieve the same trapping results.

An implementation may allow numeric signal numbers for the conditions as an extension, if and only if the following map of signal numbers to names is true:

Signal Number	Signal Name	Signal Number	Signal Name
1	SIGHUP	9	SIGKILL
2	SIGINT	14	SIGALRM
3	SIGQUIT	15	SIGTERM
6	SIGABRT		

Otherwise, it shall be an error for the application to use numeric signal numbers.

The `trap` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

If the trap name or number is invalid, a nonzero exit status shall be returned; otherwise, zero shall be returned. For both interactive and noninteractive shells, invalid signal names or numbers shall not be considered a syntax error and shall not cause the shell to abort.

3.14.13.1 `trap` Rationale. *(This subclause is not a part of P1003.2)*

Implementations may permit lowercase signal names as an extension. Implemen- 1
tations may also accept the names with the `SIG` prefix; no known historical shell 1
does so. The `trap` and `kill` utilities in POSIX.2 are now consistent in their omis- 1
sion of the `SIG` prefix for signal names. Some `kill` implementations do not allow 1
the prefix and `kill -l` lists the signals without prefixes. 1

As stated previously, when a subshell is entered, traps are set to the default 1
actions. This does not imply that the `trap` command cannot be used within the 1
subshell to set new traps. 1

Trapping SIGKILL or SIGSTOP is accepted by some historical implementations, but it does not work. Portable POSIX.2 applications cannot try it.

The output format is not historical practice. Since the output of historical `trap`s is not portable (because numeric signal values are not portable) and had to change to become so, an opportunity was taken to format the output in a way that a shell script could use to save and then later reuse a trap if it wanted. For example:

```
save_traps=$(trap)
. . .
eval "$save_traps"
```

The KornShell uses an `ERR` trap that is triggered whenever `set −e` would cause an exit. This is allowable as an extension, but was not mandated, as other shells have not used it.

The text about the environment for the `EXIT` trap invalidates the behavior of some historical versions of interactive shells which, e.g., close the standard input before executing a trap on 0. For example, in some historical interactive shell sessions the following trap on 0 would always print `--`:

```
trap 'read foo; echo "-$foo-"' 0
```

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
```

3.14.14 `unset` — Unset values and attributes of variables and functions

`unset [−fv]` *name* . . . 1

Each variable or function specified by *name* shall be unset.

If `−v` is specified, *name* refers to a variable name and the shell shall unset it and 1
remove it from the environment. Read-only variables cannot be unset. 1

If `−f` is specified, *name* refers to a function and the shell shall unset the function 1
definition. 1

If neither `−f` nor `−v` is specified, *name* refers to a variable; if a variable by that 1
name does not exist, it is unspecified whether a function by that name, if any, 1
shall be unset. 1

Unsetting a variable or function that was not previously set shall not be con-
sidered an error and shall not cause the shell to abort. 1

The `unset` special built-in shall conform to the utility argument syntax guidelines described in 2.10.2.

Exit Status

0	All *name* s were successfully unset.
>0	At least one *name* could not be unset.

3.14.14.1 `unset` Rationale. *(This subclause is not a part of P1003.2)*

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 3.5.2).

Consideration was given to omitting the `−f` option in favor of an `unfunction` utility, but decided to retain existing practice.

The `−v` option was introduced because System V historically used one name space for both variables and functions. When `unset` is used without options, System V historically unset either a function or a variable and there was no confusion about which one was intended. A portable POSIX.2 application can use `unset` without an option to unset a variable, but not a function; the `−f` option must be used.

Examples :

Unset the **VISUAL** variable:

```
unset -v VISUAL
```

Unset the functions `foo` and `bar`:

```
unset −f foo bar
```

Section 4: Execution Environment Utilities

The Execution Environment Utilities are the utilities that shall be implemented in all conforming POSIX.2 systems.

4.1 `awk` — Pattern scanning and processing language

4.1.1 Synopsis

`awk` [`−F` *ERE*] [`−v` *assignment*] . . . *program* [*argument* . . .]

`awk` [`−F` *ERE*] `−f` *progfile* . . . [`−v` *assignment*] . . . [*argument* . . .]

4.1.2 Description

The `awk` utility shall execute programs written in the *awk* programming language, which is specialized 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 shall be carried out.

Input shall be 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 shall be matched in turn against each pattern in the program. For each pattern matched, the associated action shall be executed.

The `awk` utility shall interpret 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 shall denote the first field in a record `$1`, the second `$2`, and so forth. The symbol `$0` shall refer to the entire record; setting any other field shall
cause the reevaluation of `$0`. Assigning to `$0` shall reset the values of all other 1
fields and the `NF` built-in variable. 1

4.1.3 Options

The awk utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−F *ERE* Define the input field separator to be the extended regular expression *ERE*, before any input is read (see 4.1.7.4).

−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 shall be the awk program. The awk program can alternatively be specified in the command line as a single argument.

−v *assignment*
The *assignment* argument shall be in the same form as an *assignment* operand. The specified variable assignment shall occur prior to executing the awk program, including the actions associated with BEGIN patterns (if any). Multiple occurrences of this option can be specified.

4.1.4 Operands

The following operands shall be supported by the implementation:

program If no −f option is specified, the first operand to awk shall be the text of the awk program. The application shall supply the *program* operand as a single argument to awk. If the text does not end in a <newline> character, awk shall interpret the text as if it did.

argument Either of the following two types of *argument* s 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 shall be used.

assignment
An operand that begins with an underscore or alphabetic character from the portable character set (see Table 2-3 in 2.4), followed by a sequence of underscores, digits, and alphabetics from the portable character set, followed by the = character shall specify a variable assignment rather than a pathname. The characters before the = shall represent the name of an awk variable; if that name is an awk reserved word (see 4.1.7.7) the behavior is undefined. The characters following the equals-sign shall be interpreted as if they appeared in the awk program preceded and followed by a double-

quote (") character, as a `STRING` token (see 4.1.7.7), except that if the last character is an unescaped backslash, it shall be interpreted as a literal backslash rather than as the first character of the sequence ‘\"’’. The variable shall be assigned the value of that `STRING` token. If that value is considered a *numeric string* (see 4.1.7.2), the variable shall also be assigned its numeric value. Each such variable assignment shall occur just prior to the processing of the following *file*, if any. Thus, an assignment before the first *file* argument shall be executed after the `BEGIN` actions (if any), while an assignment after the last *file* argument shall occur before the `END` actions (if any). If there are no *file* arguments, assignments shall be executed before processing the standard input.

4.1.5 External Influences

4.1.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is –. See Input Files.

4.1.5.2 Input Files

Input files to the `awk` program from any of the following sources: 1

— Any *file* operands or their equivalents, achieved by modifying the `awk` vari- 1
ables `ARGV` and `ARGC` 1

— Standard input in the absence of any *file* operands 1

— Arguments to the `getline` function 1

shall be text files. Whether the variable `RS` is set to a value other than `<new-` 1
`line>` or not, for these files, the implementation shall support records terminated 1
with the specified separator up to {LINE_MAX} bytes and may support longer 1
records. 1

If `–f` *progfile* is specified, the file(s) named by *progfile* shall be text file(s) containing an `awk` program.

4.1.5.3 Environment Variables

The following environment variables shall affect the execution of `awk`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), the behavior of character classes within regular expressions, the identification of characters as letters, and the mapping of upper- and lowercase characters for the `toupper` and `tolower` functions.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions and in comparisons of string values.

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

LC_NUMERIC This variable shall determine the radix character used when interpreting numeric input, performing conversions between numeric and string values, and formatting numeric output.

PATH This variable shall define the search path when looking for commands executed by `system(`*expr*`)`, or input and output pipes. See 2.6.

In addition, all environment variables shall be visible via the `awk` variable `ENVIRON`.

4.1.5.4 Asynchronous Events

Default.

4.1.6 External Effects

4.1.6.1 Standard Output

The nature of the output files depends on the `awk` program.

4.1.6.2 Standard Error

Used only for diagnostic messages.

4.1.6.3 Output Files

The nature of the output files depends on the `awk` program.

4.1.7 Extended Description

4.1.7.1 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 shall match any record of input, and a missing action shall be equivalent to an action that writes the matched record of input to standard output.

Execution of the `awk` program shall start 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) shall be processed in turn by reading data from the file until a record separator is seen (`<newline>` by default), splitting the current record into fields using the current value of `FS` according to the rules in 4.1.7.4, 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 shall be executed before evaluating subsequent patterns. Last, the actions associated with all `END` patterns shall be executed in the order they occur in the program.

4.1.7.2 Expressions

Expressions describe computations used in *patterns* and *actions.* In Table 4-1, 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, higher precedence operators shall 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 (i.e., on the left side of an assignment operator). The precise syntax of expressions is given in the grammar in 4.1.7.7.

Each expression shall have either a string value, a numeric value, or both. Except as stated for specific contexts, the value of an expression shall be implicitly converted to the type needed for the context in which it is used. A string value shall be converted to a numeric value by the equivalent of the following calls to functions defined by the C Standard {7}:

```
setlocale(LC_NUMERIC, "");
numeric_value = atof(string_value);
```

A numeric value that is exactly equal to the value of an integer (see 2.9.2.1) shall

Table 4-1 – awk Expressions in Decreasing Precedence

Syntax	Name	Semantic Definition	Type of Result	Assoc	
(*expr*)	Grouping	C Standard {7}	type of *expr*	n/a	
$*expr*	Field reference	4.1.7.2	string	n/a	
++ *lvalue*	Pre-increment	C Standard {7}	numeric	n/a	
−− *lvalue*	Pre-decrement	C Standard {7}	numeric	n/a	
lvalue ++	Post-increment	C Standard {7}	numeric	n/a	
lvalue −−	Post-decrement	C Standard {7}	numeric	n/a	
expr ^ *expr*	Exponentiation	4.1.7.2	numeric	right	
! *expr*	Logical not	C Standard {7}	numeric	n/a	
+ *expr*	Unary plus	C Standard {7}	numeric	n/a	
− *expr*	Unary minus	C Standard {7}	numeric	n/a	
expr * *expr*	Multiplication	C Standard {7}	numeric	left	
expr / *expr*	Division	C Standard {7}	numeric	left	
expr % *expr*	Modulus	4.1.7.2	numeric	left	
expr + *expr*	Addition	C Standard {7}	numeric	left	
expr − *expr*	Subtraction	C Standard {7}	numeric	left	
expr *expr*	String concatenation	4.1.7.2	string	left	
expr < *expr*	Less than	4.1.7.2	numeric	none	
expr <= *expr*	Less than or equal to	4.1.7.2	numeric	none	
expr != *expr*	Not equal to	4.1.7.2	numeric	none	
expr == *expr*	Equal to	4.1.7.2	numeric	none	
expr > *expr*	Greater than	4.1.7.2	numeric	none	
expr >= *expr*	Greater than or equal to	4.1.7.2	numeric	none	
expr ~ *expr*	ERE match	4.1.7.4	numeric	none	
expr !~ *expr*	ERE nonmatch	4.1.7.4	numeric	none	
expr in array	Array membership	4.1.7.2	numeric	left	
(*index*) in *array*	Multidimension array membership	4.1.7.2	numeric	left	
expr && *expr*	Logical AND	C Standard {7}	numeric	left	1
expr || *expr*	Logical OR	C Standard {7}	numeric	left	1
expr1 ? *expr2* : *expr3*	Conditional expression	C Standard {7}	type of selected *expr2* or *expr3*	right	1
lvalue ^= *expr*	Exponentiation assignment	4.1.7.2	numeric	right	
lvalue %= *expr*	Modulus assignment	4.1.7.2	numeric	right	
lvalue *= *expr*	Multiplication assignment	C Standard {7}	numeric	right	
lvalue /= *expr*	Division assignment	C Standard {7}	numeric	right	
lvalue += *expr*	Addition assignment	C Standard {7}	numeric	right	
lvalue −= *expr*	Subtraction assignment	C Standard {7}	numeric	right	
lvalue = *expr*	Assignment	C Standard {7}	type of *expr*	right	

be converted to a string by the equivalent of a call to the `sprintf` function (see 4.1.7.6.2) 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 shall 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 1
unspecified if the value of `CONVFMT` is not a floating-point format specification. 1
This standard 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 shall be considered to be a *numeric string* in the following case:

(1) Any leading and trailing `<blank>`s shall be ignored.

(2) If the first unignored character is a `+` or `–`, it shall be ignored.

(3) If the remaining unignored characters would be lexically recognized as a `NUMBER` token (as described by the lexical conventions in 4.1.7.7), the string shall be considered a *numeric string* .

If a – character is ignored in the above steps, the numeric value of the *numeric string* shall be the negation of the numeric value of the recognized `NUMBER` token. Otherwise the numeric value of the *numeric string* shall be the numeric value of the recognized `NUMBER` token. Whether or not a string is a *numeric string* shall be relevant only in contexts where that term is used in this clause.

When an expression is used in a Boolean context (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, or the expression of a `while` statement), if it has a numeric value, a value of zero shall be treated as false and any other value shall be treated as true. Otherwise, a string value of the null string shall be treated as false and any other value shall be treated as true.

All arithmetic shall follow the semantics of floating point arithmetic as specified by the C Standard {7};see 2.9.2.

The value of the expression

expr1 `^` *expr2*

shall be equivalent to the value returned by the C Standard {7}function call

`pow(`*expr1* `,` *expr2* `)`

The expression

lvalue `^=` *expr*

shall be equivalent to the C Standard {7}expression

lvalue `= pow(`*lvalue* `,` *expr* `)`

except that *lvalue* shall be evaluated only once. The value of the expression *expr1* `%` *expr2*

shall be equivalent to the value returned by the C Standard {7} function call

`fmod(`*expr1* `,` *expr2* `)`

The expression

lvalue `%=` *expr*

shall be equivalent to the C Standard {7} expression

lvalue `= fmod(`*lvalue* `,` *expr* `)`

except that *lvalue* shall be evaluated only once.

Variables and fields shall be set by the assignment statement:

lvalue `=` *expression*

and the type of *expression* shall 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 shall supply arrays that are used for storing numbers or strings. Arrays need not be declared. They shall initially be empty, and their sizes shall 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 4.1.7.7). 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 4.1.7.7). If the name used in this context is not an array name, the behavior is undefined.

A valid array *index* shall consist of one or more comma-separated expressions, similar to the way in which multidimensional arrays are indexed in some programming languages. Because `awk` arrays are really one dimensional, such a comma-separated list shall 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 shall be equivalent:

var`[`*expr1* `,` *expr2* `,` ... `,` *exprn* `]`
var`[`*expr1* `SUBSEP` *expr2* `SUBSEP` ... `SUBSEP` *exprn* `]`

A multidimensioned *index* used with the `in` operator shall be parenthesized. The `in` operator, which tests for the existence of a particular array element, shall not cause that element to exist. Any other reference to a nonexistent array element shall automatically create it.

Comparisons (with the `<`, `<=`, `!=`, `==`, `>`, and `>=` operators) shall 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 shall be converted to strings 1 as required and a string comparison shall be made using the locale-specific collation sequence. The value of the comparison expression shall be `1` if the relation is true, or `0` if the relation is false.

4.1.7.3 Variables and Special Variables

Variables can be used in an `awk` program by referencing them. With the exception of function parameters (see 4.1.7.6.2), they are not explicitly declared. Uninitialized scalar variables and array elements have both a numeric value of zero and a string value of the empty string.

Field variables shall be designated by a `$` followed by a number or numerical expression. The effect of the field number *expression* evaluating to anything other than a nonnegative integer is unspecified; uninitialized variables or string values need not be converted to numeric values in this context. New field variables can be created by assigning a value to them. References to nonexistent fields (i.e., fields after `$NF`), shall produce the null string. However, assigning to a nonexistent field [e.g., $(`NF`+2) = 5] shall 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 shall 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>`, would be considered a *numeric string* (see 4.1.7.2), the field variable shall also have the numeric value of the *numeric string* .

The implementation shall 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` shall treat the next nonnull element of `ARGV`, up through the current value of `ARGC`−1, as the name of the next input file. Thus, setting an element of `ARGV` to null means that it shall not be treated as an input file. The name `'-'` shall indicate the standard input. If an argument matches the format of an *assignment* operand, this argument shall 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 POSIX.1 {8} 2.7. The indices of the array shall be strings consisting of the names of the environment variables, and the value of each array element shall be a string consisting of the value of that variable. If the value of an environment variable is considered a *numeric string* (see 4.1.7.2), the array element shall also have its numeric value.

In all cases where the behavior of `awk` is affected by environment variables [including the environment of any command(s) 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 shall be the environment at the time `awk` began executing; it is implementation defined 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; `<space>` 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` shall 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 4.1.7.6.1); `"%.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; `<space>` by default.

`ORS` The `print` statement output record separator; `<newline>` 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; `<newline>` 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 `<newline>` 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 multidimensional arrays; the default value is implementation defined.

4.1.7.4 Regular Expressions

The `awk` utility shall make use of the extended regular expression notation (see 2.8.4) except that it shall allow the use of C-language conventions for escaping special characters within the EREs, as specified in Table 2-15 and Table 4-2; these escape sequences shall be recognized both inside and outside bracket expressions. Note that records need not be separated by `<newline>`s and string constants can contain `<newline>`s, so even the `\n` sequence is valid in `awk` EREs. Using a slash character within the regular expression requires the escaping shown in Table 4-2.

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 shall 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 shall evaluate to a value of `1`, and the `!~` expression shall evaluate to a value of `0`. (The regular expression matching operation is as defined in 2.8.1.2, 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 shall evaluate to a value of `0`, and the `!~` expression shall 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 shall be interpreted as an extended regular expression, including the escape conventions described above. Note that these same escape conventions also shall be applied in the determining the value of a string literal (the lexical token `STRING`), and thus shall 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 shall be the equivalent of

`$0 ~ /`*ere*`/`

The *ERE* argument to the `gsub`, `match`, `sub` functions, and the *fs* argument to the `split` function (see 4.1.7.6.2) shall be interpreted as extended regular expressions. These can be either `ERE` tokens or arbitrary expressions, and shall 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 shall be a single `<space>` character. The following describes `FS` behavior:

(1) If FS is a single character:

(a) If FS is <space>, skip leading and trailing <blank>s; fields shall be delimited by sets of one or more <blank>s.

(b) Otherwise, if FS is any other character *c*, fields shall be delimited by each single occurrence of *c*.

(2) Otherwise, the string value of FS shall be considered to be an extended regular expression. Each occurrence of a sequence matching the extended regular expression shall delimit fields.

Except in the gsub, match, split, and sub built-in functions, regular expression matching shall be based on input records; i.e., record separator characters (the first character of the value of the variable RS, <newline> by default) cannot be embedded in the expression, and no expression shall match the record separator character. If the record separator is not <newline>, <newline> characters embedded in the expression can be matched. In those four built-in functions, regular expression matching shall be based on text strings; i.e., any character (including <newline> and the record separator) can be embedded in the pattern and an appropriate pattern shall 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.

4.1.7.5 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.

4.1.7.5.1 Special Patterns

The awk utility shall recognize two special patterns, BEGIN and END. Each BEGIN pattern shall 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 4.1.7.6.2) in a prior BEGIN action] and before command line assignment is done. Each END pattern shall be matched once and its associated action executed after the last record of input has been read. These two patterns shall have associated actions.

BEGIN and END shall not combine with other patterns. Multiple BEGIN and END patterns shall be allowed. The actions associated with the BEGIN patterns shall 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 shall 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 shall be read before the statements in the END action(s) are executed.

4.1.7.5.2 Expression Patterns

An expression pattern shall be evaluated as if it were an expression in a Boolean 1
context. If the result is true, the pattern shall be considered to match, and the 1
associated action (if any) shall be executed. If the result is false, the action shall 1
not be executed. 1

4.1.7.5.3 Pattern Ranges

A pattern range consists of two expressions separated by a comma; in this case, the action shall 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.

4.1.7.6 Actions

An action is a sequence of statements as shown in the grammar in 4.1.7.7. Any single statement can be replaced by a statement list enclosed in braces. The statements in a statement list shall be separated by `<newline>`s or semicolons, and shall be executed sequentially in the order that they appear.

The *expression* acting as the conditional in an `if` statement shall be evaluated and if it is nonzero or nonnull, the following *statement* shall be executed; otherwise, if `else` is present, the statement following the `else` shall be executed.

The `if`, `while`, `do` ... `while`, `for`, `break`, and `continue` statements are based on the C Standard {7} (see 2.9.2), except that the Boolean expressions shall be treated as described in 4.1.7.2, and except in the case of

`for (`*variable* `in` *array* `)`

which shall 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 behavior is undefined.

The `delete` statement shall remove an individual array element. Thus, the following code shall delete an entire array:

```
for (index in array)
        delete array[index]
```

The `next` statement shall cause all further processing of the current input record to be abandoned. The behavior is undefined if a `next` statement appears or is invoked in a `BEGIN` or `END` action.

The `exit` statement shall 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 shall terminate the program without further execution of `END` actions. If an expression is specified in an `exit` statement, its numeric value shall be the exit status of `awk`, unless subsequent errors are encountered or a subsequent `exit` statement with an expression is executed.

4.1.7.6.1 Output Statements

Both `print` and `printf` statements shall write to standard output by default. The output shall be written to the location specified by *output_redirection* if one is supplied, as follows:

> *expression*
>> *expression*
| *expression*

In all cases, the *expression* shall 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 shall be opened, creating it if necessary, and using the first form, truncating the file. The output then shall be appended to the file. As long as the file remains open, subsequent calls in which *expression* evaluates to the same string value simply shall append output to the file. The file remains open until the `close` function (see 4.1.7.6.2). is called with an expression that evaluates to the same string value.

The third form shall write output onto a stream piped to the input of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall be equivalent to one created by a call to the *popen* () function (see B.3.2) 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 shall write output to the existing stream. The stream shall remain open until the `close` function (see 4.1.7.6.2) is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose* () function (see B.3.2).

As described in detail by the grammar in 4.1.7.7, these output statements shall 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 shall 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 shall be taken as strings, being converted if
necessary; this conversion shall be as described in 4.1.7.2, with the exception that 1
the `printf` format in `OFMT` shall be used instead of the value in `CONVFMT`. An 1
empty expression list shall stand for the whole input record (`$0`).

The `printf` statement shall produce output based on a notation similar to the File Format Notation used to describe file formats in this standard (see 2.12). Output shall be produced as specified with the first expression argument as the string <*format*> and subsequent expression arguments as the strings <*arg1*> through <*argn*>, with the following exceptions:

(1) The *format* shall 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, shall 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 shall be treated as an ordinary character that is copied to the output.

(3) The *escape sequences* beginning with a backslash character shall be treated as sequences of ordinary characters that are copied to the output. (Note that these same sequences shall be interpreted lexically by `awk` when they appear in literal strings, but they shall 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 shall be fetched and its numeric value taken as the field width or precision.

(5) The implementation shall not precede or follow output from the `d` or `u` conversion specifications with `<blank>`s not specified by the *format* string.

(6) The implementation shall not precede output from the `o` conversion specification with leading zeroes 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 shall be output. If the value is zero or is not the encoding of any character in the character set, the behavior is undefined. If the argument does not have a numeric value, the first character of the string value shall be output; if the string does not contain any characters the behavior is undefined.

(8) For each conversion specification that consumes an argument, the next expression argument shall be evaluated. With the exception of the `c` conversion, the value shall be converted (according to the rules specified in 4.1.7.2) 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 behavior is undefined.

(10) If any character sequence in the *format* string begins with a % character, but does not form a valid conversion specification, the behavior is unspecified.

Both `print` and `printf` can output at least {LINE_MAX} bytes.

4.1.7.6.2 Functions

The `awk` language has a variety of built-in functions: arithmetic, string, input/output, and general.

4.1.7.6.2.1 Arithmetic Functions

The arithmetic functions, except for `int`, shall be based on the C Standard {7}; see 2.9.2. The behavior is undefined in cases where the C Standard {7} specifies that an error be returned or that the behavior 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 shall be truncated toward 0 when $x > 0$.

`rand()` Return a random number n, such that $0 \leq n < 1$.

`srand([expr])` Set the seed value for `rand` to *expr* or use the time of day if *expr* is omitted. The previous seed value shall be returned.

4.1.7.6.2.2 String Functions

The string functions are:

`gsub(`*ere*`,` *repl*`[,` *in*`])`
Behave like `sub` (see below), except that it shall 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 record, `$0`, if there is no argument.

`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 shall be set to the starting position (which is the same as the returned value), zero if no match is found; RLENGTH shall 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 returns *n*. The separation shall be done with the extended regular expression *fs* or with the field separator

`FS` if *fs* is not given. Each array element shall 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>`, would be considered a *numeric string* (see 4.1.7.2), the array element shall 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* shall be replaced by the string from *in* that matches the regular expression. An ampersand preceded by a backslash within *repl* shall be interpreted as a literal ampersand character. If *in* is specified and it is not an *lvalue* (see 4.1.7.2), the behavior is undefined. If *in* is omitted, `awk` shall 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 shall 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 uppercase letter specified to have a `tolower` mapping by the LC_CTYPE category of the current locale shall be replaced in the returned string by the lowercase letter specified by the mapping. Other characters in *s* shall be unchanged in the returned string.

`toupper(`*s*`)` Return a string based on the string *s*. Each character in *s* that is a lowercase letter specified to have a `toupper` mapping by the LC_CTYPE category of the current locale shall be replaced in the returned string by the uppercase letter specified by the mapping. Other characters in *s* shall 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 4.1.7.4.

4.1.7.6.2.3 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 defined. If the close was successful, the function shall return zero; otherwise, it shall return nonzero.

expression `| getline` [*var*]
: Read a record of input from a stream piped from the output of a command. The stream shall be created if no stream is currently open with the value of *expression* as its command name. The stream created shall 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 shall read subsequent records from the file. The stream shall remain open until the `close` function is called with an expression that evaluates to the same string value. At that time, the stream shall be closed as if by a call to the *pclose*() function. If *var* is missing, `$0` and `NF` shall be set; otherwise, *var* shall be set.

`getline` Set `$0` to the next input record from the current input file. This form of `getline` shall 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` shall set the `FNR` and `NR` variables.

`getline` [*var*] `<` *expression*
: Read the next record of input from a named file. The *expression* shall be evaluated to produce a string that is used as a full pathname. If the file of that name is not currently open, it shall be opened. As long as the stream remains open, subsequent calls in which *expression* evaluates to the same string value shall read subsequent records from the file. The file shall 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` shall be set; otherwise, *var* shall be set.

`system(`*expression*`)`
: Execute the command given by *expression* in a manner equivalent to the *system*() function [see B.3.1] and return

the exit status of the command.

All forms of `getline` shall return 1 for successful input, zero for end of file, and −1 for an error.

4.1.7.6.2.4 User-Defined Functions

The `awk` language also shall provide 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 shall be global.

Function arguments can be either scalars or arrays; the behavior 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 shall be passed by value if scalar and by reference if array name. Argument names shall be local to the function; all other variable names shall be global. The same name shall not be used as both an argument name and as the name of a function or a special `awk` variable. The same name shall not be used both as a variable name with global scope and as the name of a function. The same name shall 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 shall be initialized 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 shall be initialized as empty arrays. If more arguments are supplied in a function call than are in the function definition, the behavior is undefined.

When invoking a function, no white space can be placed between the function name and the opening parenthesis. The implementation shall permit function calls to be nested, and for recursive calls to be made upon functions. Upon return from any nested or recursive function call, the values of all of the calling function's parameters shall 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 behavior is undefined.

In the function definition, `<newline>`s shall be 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.

4.1.7.7 `awk` Grammar

The grammar in this subclause and the lexical conventions in the following subclause shall together describe the syntax for `awk` programs. The general conventions for this style of grammar are described in 2.1.2. A valid program can be represented as the nonterminal symbol *program* in the grammar. Any discrepancies found between this grammar and other descriptions in this clause shall be resolved in favor of this grammar.

```
%token   NAME NUMBER STRING ERE NEWLINE
%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   '{' '}' '(' ')' '[' ']' ',' ';'
%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:
          ';'
        | NEWLINE
        | terminator NEWLINE ';'                                        2
        ;

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
          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 */
        | 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 */
        | 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 shall be resolved as follows:

— Operator precedence and associativity shall be as described in Table 4-1.

— In case of ambiguity, an `else` shall be associated with the most immediately preceding `if` that would satisfy the grammar.

4.1.7.8 awk Lexical Conventions

The lexical conventions for `awk` programs, with respect to the preceding grammar, shall be as follows:

(1) Except as noted, `awk` shall recognize the longest possible token or delimiter beginning at a given point.

(2) A comment shall consist of any characters beginning with the number sign character and terminated by, but excluding the next occurrence of, a `<newline>` character. Comments shall have no effect, except to delimit lexical tokens.

(3) The character `<newline>` shall be recognized as the token `NEWLINE`.

(4) A backslash character immediately followed by a `<newline>` character 1
shall have no effect. 1

(5) The token `STRING` shall represent a string constant. A string constant shall begin with the character `"`. Within a string constant, a backslash character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape sequences in Table 4-2 shall be recognized. A `<newline>` character shall not occur within a string constant. A string constant shall be terminated by the first unescaped occurrence of the character `"` after the one that begins the string constant. The value of the string shall 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 shall begin with the slash character. Within an ERE constant, a `<backslash>` character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape 1
sequences in Table 4-2 shall be recognized. A `<newline>` character shall not occur within an ERE constant. An ERE constant shall 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 shall be the sequence of all unescaped characters and values of escape sequences between, but not including, the two delimiting slash characters.

(7) A `<blank>` shall have no effect, except to delimit lexical tokens or within `STRING` or `ERE` tokens.

(8) The token `NUMBER` shall represent a numeric constant. Its form and numeric value shall be equivalent to the either of the tokens `floating-constant` or `integer-constant` as specified by the C Standard {7}, with the following exceptions:

Table 4-2 – awk Escape Sequences

Escape Sequence	Description	Meaning	
`\"`	`<backslash>` `<quotation-mark>`	`<quotation-mark>` character	
`\/`	`<backslash>` `<slash>`	`<slash>` character	
ddd	`<backslash>` followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (i.e., representation of the NUL character), the behavior 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 defined. Multibyte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.	1
c	`<backslash>` followed by any character not described in this table or in Table 2-15	Undefined	

(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` shall 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 (see 2.9.2.1), the behavior is undefined.

(9) A sequence of underscores, digits, and alphabetics from the portable character set (see 2.4), beginning with an underscore or alphabetic, shall be considered a word.

(10) The following words are keywords that shall be recognized 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 shall be recognized 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 shall consist 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 shall consist 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 shall not be included as part of the token.

(14) The following two-character sequences shall be recognized 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 shall be recognized 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 recognized shall be recognized. In any other syntactic context where the token ERE could appear as the next token in a valid program, the token ERE shall be recognized.

4.1.8 Exit Status

The awk utility shall exit with one of the following values:

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.

4.1.9 Consequences of Errors

If any *file* operand is specified and the named file cannot be accessed, `awk` shall write a diagnostic message to standard error and terminate without any further action.

If the program specified by either the *program* operand or the *progfile* operand(s) is not a valid `awk` program (as specified in 4.1.7), the behavior is undefined.

4.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `awk` program specified in the command line is most easily specified within single-quotes (e.g., '*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) Write any line in which the second field matches the regular expression and the fourth field does not.

```
$2 ~ /xyz/ && $4 !~ /xyz/
```

(5) Write any line in which the second field contains a backslash.

```
$2 ~ /\\/
```

(6) 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 ~ "\\\\"
```

(7) Write the second to the last and the last field in each line. Separate the fields by a colon.

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

(8) 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}
```

(9) Write lines longer than 72 characters.

```
{length($0) > 72}
```

(10) Write first two fields in opposite order separated by the OFS:

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

(11) Same, with input fields separated by comma and/or <space>s and <tab>s:

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

(12) Add up first column, print sum and average.

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

(13) Write fields in reverse order, one per line (many lines out for each line in):

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

(14) Write all lines between occurrences of the strings start and stop:

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

(15) Write all lines whose first field is different from the previous one:

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

(16) Simulate echo:

```
BEGIN  {
       for (i = 1; i < ARGC; ++i)
              printf "%s%s", ARGV[i], i==ARGC-1?"\n":""
}
```

(17) 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]
}
```

(18) 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.

The `index`, `length`, `match`, and `substr` should not be confused with similar functions in the C Standard {7}; the `awk` versions deal with characters, while the C Standard {7} deals with bytes.

To forestall any possible confusion, where strings are used as the name of a file or 1
pipeline, the strings must be textually identical. The terminology ‘‘same string 1
value’’ implies that ‘‘equivalent strings,’’ even those that differ only by `<space>`s, 1
represent different files. 1

History of Decisions Made

This description is based on the new `awk`, ‘‘`nawk`,’’ (see *The AWK Programming Language* {B21}), which introduced a number of new features to the historical `awk`:

(1) New keywords: `delete`, `do`, `function`, `return`

(2) New built-in functions: `atan2`, `cos`, `sin`, `rand`, `srand`, `gsub`, `sub`, `match`, `close`, `system`

(3) New predefined variables: `FNR`, `ARGC`, `ARGV`, `RSTART`, `RLENGTH`, `SUBSEP`

(4) New expression operators: `?:`, `^`

(5) The `FS` variable and the third argument to `split` are now treated as extended regular expressions.

(6) The operator precedence has changed to more closely match C. Two examples of code that operate differently are:

```
while ( n /= 10 > 1) ...
if (!"wk" ~ /bwk/) ...
```

Several features have been added based on newer implementations of `awk`:

(1) Multiple instances of `−f` *progfile* are permitted.

(2) New option: `−v` *assignment*

(3) New predefined variable: `ENVIRON`

(4) New built-in functions: `toupper`, `tolower`

(5) More formatting capabilities added to `printf` to match the C Standard {7}.

Regular expressions have been extended somewhat from traditional implementations to make them a pure superset of Extended Regular Expressions as defined by this standard (see 2.8.4). The main extensions are internationalization features and interval expressions. Traditional implementations of awk have long supported <backslash> escape sequences as an extension to regular expressions, and this extension has been retained despite inconsistency with other utilities. The number of escape sequences recognized in both regular expressions and strings has varied (generally increasing with time) among implementations. The set specified by the standard includes most sequences known to be supported by popular implementations and by the C Standard {7}. One sequence that is not supported is hexadecimal value escapes beginning with "\x". This would allow values expressed in more than 9 bits to be used within awk as in the C Standard {7}. However, because this syntax has a nondeterministic length, it does not permit the subsequent character to be a hexadecimal digit. This limitation can be worked around in the C language by the use of lexical string concatenation. In the awk language, concatenation could also be a solution for strings, but not for regular expressions (either lexical ERE tokens or strings used dynamically as regular expressions). Because of this limitation, the feature has not been added to POSIX. 2.

When a string variable is used in a context where an ERE normally appears 1
(where the lexical token ERE is used in the grammar) the string does not contain 1
the literal slashes. 1

Some versions of awk allow the form:

func *name* (*args* , . . .) { *statements* }

This has been deprecated by the language's authors, who have asked that it not be included in the standard.

Traditional implementations of awk produce an error if a next statement is executed in a BEGIN action, and cause awk to terminate if a next statement is executed in an END action. This behavior has not been documented, and it was not believed that it was necessary to standardize it.

The specification of conversions between string and numeric values is much more detailed than in the documentation of traditional implementations or in *The AWK Programming Language* {B21}. Although most of the behavior is designed to be intuitive, the details are necessary to ensure compatible behavior from different implementations. This is especially important in relational expressions, since the types of the operands determine whether a string or numeric comparison is performed. From the perspective of an application writer, it is usually sufficient to expect intuitive behavior and to force conversions (by adding zero or concatenating a null string) when the type of an expression does not obviously match what is needed. The intent has been to specify existing practice in almost all cases. The one exception is that, in traditional implementations, variables and constants maintain both string and numeric values after their original value is converted by any use. This means that referencing a variable or constant can have unexpected side effects. For example, with traditional implementations the following program:

```
{
      a = "+2"
      b = 2
      if (NR % 2)
            c = a + b
      if (a == b)
            print "numeric comparison"
      else
            print "string comparison"
}
```

would perform a numeric comparison (and output **numeric comparison**) for each odd-numbered line, but perform a string comparison (and output **string comparison**) for each even-numbered line. POSIX.2 ensures that comparisons will be numeric if necessary. With traditional implementations, the following program:

```
BEGIN {
        OFMT = "%e"
        print 3.14
        OFMT = "%f"
        print 3.14
}
```

would output **3.140000e+00** twice, because in the second `print` statement the constant `3.14` would have a string value from the previous conversion. The standard requires that the output of the second `print` statement be **3.140000**. The behavior of traditional implementations was seen as too unintuitive and unpredictable.

However, a further modification was made in Draft 11. It was pointed out that with the Draft 10 rules, the following script would print nothing:

```
BEGIN {
        y[1.5] = 1
        OFMT = "%e"
        print y[1.5]
}
```

Therefore, a new variable, `CONVFMT`, was introduced. The `OFMT` variable is now restricted to affecting output conversions of numbers to strings and `CONVFMT` is used for internal conversions, such as comparisons or array indexing. The default value is the same as that for `OFMT`, so unless a program changes `CONVFMT` (which no historical program would do), it will receive the historical behavior associated with internal string conversions.

The POSIX `awk` lexical and syntactic conventions are specified more formally than in other sources. Again the intent has been to specify existing practice. One convention that may not be obvious from the formal grammar as in other verbal descriptions is where `<newline>`s are acceptable. There are several obvious placements such as terminating a statement, and a backslash can be used to escape `<newline>`s between any lexical tokens. In addition, `<newline>`s 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 }
```

The requirement that `awk` add a trailing `<newline>` to the *program* argument text is to simplify the grammar, making it match a text file in form. There is no way for an application or test suite to determine whether a literal `<newline>` is added or whether `awk` simply acts as if it did.

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.

The overall `awk` syntax has always been based on the C language, with a few features from the shell command language and other sources. Because of this, it is not completely compatible with any other language, which has caused confusion for some users. It is not the intent of this standard to address such issues. The standard has made a few relatively minor changes toward making the language more compatible with the C language as specified by the C Standard {7}; most of these changes are based on similar changes in recent implementations, as described above. There remain several C language conventions that are not in *awk* . One of the notable ones is the comma operator, which is commonly used to specify multiple expressions in the C language `for` statement. Also, there are various places where `awk` is more restrictive than the C language regarding the type of expression that can be used in a given context. These limitations are due to the different features that the `awk` language does provide.

This standard requires several changes from traditional implementations in order to support internationalization. Probably the most subtle of these is the use of the decimal-point character, defined by the LC_NUMERIC category of the locale, in representations of floating point numbers. This locale-specific character is used in recognizing numeric input, in converting between strings and numeric values, and in formatting output. However, regardless of locale, the period character (the decimal-point character of the POSIX Locale) is the decimal-point character recognized in processing `awk` programs (including assignments in command-line arguments). This is essentially the same convention as the one used in the C Standard {7}. The difference is that the C language includes the *setlocale* () function, which permits an application to modify its locale. Because of this capability, a C application begins executing with its locale set to the C locale, and only executes in the environment-specified locale after an explicit call to *setlocale* (). However, adding such an elaborate new feature to the `awk` language was seen as inappropriate for POSIX. 2. It is possible to explicitly execute an `awk` program in any desired locale by setting the environment in the shell.

The behavior in the case of invalid `awk` programs (including lexical, syntactic, and semantic errors) is undefined because it was considered overly limiting on implementations to specify. In most cases such errors can be expected to produce a diagnostic and a nonzero exit status. However, some implementations may choose to extend the language in ways that make use of certain invalid constructs.

Other invalid constructs might be deemed worthy of a warning but otherwise cause some reasonable behavior. Still other constructs may be very difficult to detect in some implementations. Also, different implementations might detect a given error during an initial parsing of the program (before reading any input files) while others might detect it when executing the program after reading some input. Implementors should be aware that diagnosing errors as early as possible and producing useful diagnostics can ease debugging of applications, and thus make an implementation more usable.

The unspecified behavior from using multicharacter `RS` values is to allow possible future extensions based on regular expressions used for record separators. Historical implementations take the first character of the string and ignore the others.

The undefined behavior resulting from NULs in regular expressions allows future extensions for the GNU `gawk` program to process binary data.

Unspecified behavior when `split(string,array,<null>)` is used is to allow a proposed future extension that would split up a string into an array of individual characters.

4.2 `basename` — Return nondirectory portion of pathname

4.2.1 Synopsis

`basename` *string* [*suffix*]

4.2.2 Description

The *string* operand shall be treated as a pathname, as defined in 2.2.2.102. The string *string* shall be converted to the filename corresponding to the last pathname component in *string* and then the suffix string *suffix* , if present, shall be removed. This shall be done by performing actions equivalent to the following steps in order:

(1) If *string* is `//`, it is implementation defined whether steps (2) through (5) are skipped or processed.

(2) If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps (3) through (5).

(3) If there are any trailing slash characters in *string* , they shall 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* shall 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* shall be removed from *string* . Otherwise, *string*

shall not be modified by this step. It shall not be considered an error if *suffix* is not found in *string* .

The resulting string shall be written to standard output.

4.2.3 Options

None.

4.2.4 Operands

The following operands shall be supported by the implementation:

string A string.

suffix A string.

4.2.5 External Influences

4.2.5.1 Standard Input

None.

4.2.5.2 Input Files

None.

4.2.5.3 Environment Variables

The following environment variables shall affect the execution of `basename`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.2.5.4 Asynchronous Events

Default.

4.2.6 External Effects

4.2.6.1 Standard Output

The `basename` utility shall write a line to the standard output in the following format:

`"%s\n"`, <*resulting string*>

4.2.6.2 Standard Error

Used only for diagnostic messages.

4.2.6.3 Output Files

None.

4.2.7 Extended Description

None.

4.2.8 Exit Status

The `basename` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

4.2.9 Consequences of Errors

Default.

4.2.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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)
```

History of Decisions Made

The POSIX.1 {8} definition of pathname allows trailing slashes on a pathname naming a directory. Some historical implementations have not allowed trailing slashes and thus treated pathnames of this form in other ways. Existing implementations also differ in their handling of *suffix* when *suffix* matches the entire string left after removing the directory part of *string*.

The behaviors of `basename` and `dirname` in this standard have been coordinated so that when *string* is a valid pathname

```
$(basename "string")
```

would be a valid filename for the file in the directory

```
$(dirname "string")
```

This would not work for the versions of these utilities in earlier drafts due to the way it specified handling of trailing slashes.

Since the definition of *pathname* in 2.2.2.102 specifies implementation-defined behavior for pathnames starting with two slash characters, Draft 11 has been changed to specify similar implementation-defined behavior for the `basename` and `dirname` utilities. On implementations where the pathname `//` is always treated the same as the pathname `/`, the functionality required by Draft 10 meets all of the Draft 11 requirements.

4.3 `bc` — Arbitrary-precision arithmetic language

4.3.1 Synopsis

`bc [−l]` [*file* . . .]

4.3.2 Description

The `bc` utility shall implement an arbitrary precision calculator. It shall take input from any files given, then read from the standard input. If the standard input and standard output to `bc` are attached to a terminal, the invocation of `bc` shall be considered to be *interactive*, causing behavioral constraints described in the following subclauses.

4.3.3 Options

The `bc` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−l` (The letter ell.) Define the math functions and initialize `scale` to 20, instead of the default zero. See 4.3.7.

4.3.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a text file containing `bc` program statements. After all *file*s have been read, `bc` shall read the standard input.

4.3.5 External Influences

4.3.5.1 Standard Input

See Input Files.

4.3.5.2 Input Files

Input files shall be text files containing a sequence of comments, statements, and function definitions that shall be executed as they are read.

4.3.5.3 Environment Variables

The following environment variables shall affect the execution of `bc`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.3.5.4 Asynchronous Events

Default.

4.3.6 External Effects

4.3.6.1 Standard Output

The output of the `bc` utility shall be controlled by the program read, and shall consist of zero or more lines containing the value of all executed expressions 2
without assignments. The radix and precision of the output shall be controlled by the values of the `obase` and `scale` variables. See 4.3.7.

4.3.6.2 Standard Error

Used only for diagnostic messages.

4.3.6.3 Output Files

None.

4.3.7 Extended Description

4.3.7.1 bc Grammar

The grammar in this subclause and the lexical conventions in the following subclause shall together describe the syntax for bc programs. The general conventions for this style of grammar are described in 2.1.2. A valid program can be represented as the nonterminal symbol program in the grammar. Any discrepancies found between this grammar and other descriptions in this subclause (4.3.7) shall be resolved in favor of this grammar.

```
%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                  1
```

```
                       | 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
                       ;
```

4.3.7.2 bc Lexical Conventions

The lexical conventions for `bc` programs, with respect to the preceding grammar, shall be as follows:

(1) Except as noted, `bc` shall recognize the longest possible token or delimiter beginning at a given point.

(2) A comment shall consist of any characters beginning with the two adjacent characters /* and terminated by the next occurrence of the two adjacent characters */. Comments shall have no effect except to delimit lexical tokens.

(3) The character `<newline>` shall be recognized as the token `NEWLINE`.

(4) The token `STRING` shall represent a string constant; it shall consist of any characters beginning with the double-quote character (`"`) and terminated by another occurrence of the double-quote character. The value of the string shall be the sequence of all characters between, but not including, the two double-quote characters. All characters shall be 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 shall be limited to {BC_STRING_MAX} bytes.

(5) A `<blank>` shall have 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 shall delimit lexical tokens with the following exceptions:

— It shall be interpreted as a literal `<newline>` in `STRING` tokens.

— It shall be ignored as part of a multiline `NUMBER` token.

(7) The token `NUMBER` shall represent a numeric constant. It shall be recognized 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 shall be interpreted as a numeral in the base specified by the value of the internal register ibase (described below). Each of the digit characters shall have the value from 0 to 15 in the order listed here, and the period character shall represent the radix point. The behavior 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 4.3.7.3).

(9) The following keywords shall be recognized 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 shall be recognized 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 shall be recognized 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 behavior is undefined.

(13) The following single-characters shall be recognized as the token MUL_OP:

* / %

(14) The following single-character and two-character sequences shall be recognized as the token REL_OP:

== <= >= != < >

(15) The following two-character sequences shall be recognized as the token INCR_DECR:

++ --

(16) The following single characters shall be recognized as tokens whose names are the character:

`<newline>   (   )   ,   +   -   ;   [   ]   ^   {   }` 1

(17) The token `EOF` shall be returned when the end of input is reached.

4.3.7.3 `bc` Operations

There are three kinds of identifiers: ordinary identifiers, array identifiers, and function identifiers. All three types consist of single lowercase letters. Array identifiers shall be 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 shall be truncated to integers. Function identifiers shall be followed by parentheses, possibly enclosing arguments. The three types of identifiers do not conflict.

Table 4-3 summarizes the rules for precedence and associativity of all operators. Operators on the same line shall have the same precedence; rows are in order of decreasing precedence.

Table 4-3 – `bc` Operators

Operator	Associativity
++, −−	not applicable
unary −	not applicable
^	right to left
*, /, %	left to right
+, binary −	left to right
=, +=, −=, *=, /=, %=, ^=	right to left
==, <=, >=, !=, <, >	none

Each expression or named expression has a *scale*, which is the number of decimal digits that shall be maintained as the fractional portion of the expression.

Named expressions are places where values are stored. Named expressions shall be valid on the left side of an assignment. The value of a named expression shall be the value stored in the place named. Simple identifiers and array elements shall be named expressions; they shall 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 shall be zero; values assigned to any of these registers shall be truncated to integers. The `scale` register shall contain a global value used in computing the scale of expressions (as described below). The value of the register `scale` shall be limited to 0 ≤ `scale` ≤ {BC_SCALE_MAX} and shall have a default value of zero. The `ibase` and `obase` registers are the input and output number radix, respectively. The value of `ibase` shall be limited to

2 ≤ `ibase` ≤ 16

The value of `obase` shall be limited to

2 ≤ `obase` ≤ {BC_BASE_MAX}

When either `ibase` or `obase` is assigned a single `digit` value from the list in 4.3.7.2, the value shall be assumed in hexadecimal. (For example, `ibase=A` sets to base ten, regardless of the current `ibase` value.) Otherwise, the behavior is undefined when digits greater than or equal to the value of `ibase` appear in the input. Both `ibase` and `obase` shall have initial values of 10.

Internal computations shall be conducted as if in decimal, regardless of the input 1
and output bases, to the specified number of decimal digits. When an exact result is not achieved, (e.g., `scale=0; 3.2/1`) the result shall be truncated.

For all values of `obase` specified by this standard, numerical values shall be output as follows:

(1) If the value is less than zero, a hyphen (`-`) character shall be output.

(2) One of the following shall 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 shall be output as a series of digits appropriate to `obase` (as described below). The most significant nonzero digit shall 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 shall be output.

(3) If the scale of the value is greater than zero, a period character shall 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 shall 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 through fifteen, respectively.

For bases greater than 16, each ‘‘digit’’ shall be written as a separate multidigit decimal number. Each digit except the most significant fractional digit shall be preceded a single <space> character. For bases from 17 to 100, `bc` shall write two-digit decimal numbers; for bases from 101 to 999, 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 shall 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 shall end with a backslash (\).

A function call shall consist 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 shall be specified by the array name followed by empty square brackets. All function arguments shall be 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 shall be the value of the expression in the parentheses of the `return` statement or shall be zero if no expression is provided or if there is no `return` statement.

The result of `sqrt`(*expression*) shall be the square root of the expression. The result shall be truncated in the least significant decimal place. The scale of the result shall be the scale of the expression or the value of `scale`, whichever is larger.

The result of `length`(*expression*) shall be the total number of significant decimal digits in the expression. The scale of the result shall be zero.

The result of `scale`(*expression*) shall be the scale of the expression. The scale of the result shall be zero.

A numeric constant shall be an expression. The scale shall 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* `)` shall 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 shall be the negative of the *expression* . The scale of the result shall be the scale of *expression* .

The unary increment and decrement operators shall not modify the scale of the named expression upon which they operate. The scale of the result shall be the scale of that named expression.

++*named-expression*
: The named expression shall be incremented by one. The result shall be the value of the named expression after incrementing.

– –*named-expression*
: The named expression shall be decremented by one. The result shall be the value of the named expression after decrementing.

named-expression ++
The named expression shall be incremented by one. The result shall be the value of the named expression before incrementing.

named-expression −−
The named expression shall be decremented by one. The result shall be the value of the named expression before decrementing.

The exponentiation operator, circumflex (^), shall bind right to left.

expression ^ *expression*
The result shall be the first *expression* raised to the power of the second *expression*. If the second expression is not an integer, the behavior 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 shall be:

```
if b >= 0 min(a * b, max(scale, a))   2
if b <  0 scale                        2
```

The multiplicative operators (*, /, %) shall bind left to right.

expression * *expression*
The result shall be the product of the two expressions. If `a` and `b` are the scales of the two expressions, then the scale of the result shall be:

```
min(a+b,max(scale,a,b))
```

expression / *expression*
The result shall be the quotient of the two expressions. The scale of the result shall be the value of `scale`.

expression % *expression*
For expressions *a* and *b*, `a % b` shall 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 shall be

```
max(scale + scale(b), scale(a))
```

The additive operators (+, −) shall bind left to right.

expression + *expression*
The result shall be the sum of the two expressions. The scale of the result shall be the maximum of the scales of the expressions.

expression − *expression*
The result shall be the difference of the two expressions. The scale of

the result shall be the maximum of the scales of the expressions.

The assignment operators (=, +=, −=, *=, /=, %=, ^=) shall 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 shall be the scale of *expression*.

The compound assignments forms

named-expression *<operator>*= *expression*

shall be equivalent to:

named-expression = *named-expression* *<operator>* *expression*

except that the *named-expression* shall be evaluated only once.

Unlike all other operators, the relational operators (<, >, <=, >=, ==, !=) shall be only valid as the object of an `if`, `while`, or inside a `for` statement.

expression1 < *expression2*
: The relation shall be true if the value of *expression1* is strictly less than the value of *expression2*.

expression1 > *expression2*
: The relation shall be true if the value of *expression1* is strictly greater than the value of *expression2*.

expression1 <= *expression2*
: The relation shall be true if the value of *expression1* is less than or equal to the value of *expression2*.

expression1 >= *expression2*
: The relation shall be true if the value of *expression1* is greater than or equal to the value of *expression2*.

expression1 == *expression2*
: The relation shall be true if the values of *expression1* and *expression2* are equal.

expression1 != *expression2*
: The relation shall 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 shall 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 shall 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 shall 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 return is made from the function, 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 shall write the value of the expression followed by a `<newline>` character.

When a statement is a string, execution of the statement shall write the value of the string.

Statements separated by semicolon or `<newline>` shall 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` shall be executed immediately and any output produced by that execution shall be written without any delay due to buffering.

In an `if` statement [`if` (*relation*) *statement*] the *statement* shall 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* shall be executed and the *relation* retested. When the *relation* is false, execution shall resume after *statement* .

A `for` statement [`for` (*expression* ; *relation*; *expression*) *statement*] shall be the same as:

```
first-expression
while (relation ) {
        statement
        last-expression
}
```

All three expressions shall be present.

The `break` statement causes termination of a `for` or `while` statement.

The `auto` statement [`auto` *identifier* [,*identifier*] . . .] shall cause the values of the identifiers to be pushed down. The identifiers can be ordinary identifiers or array identifiers. Array identifiers shall be specified by following the array name by empty square brackets. The `auto` statement shall 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 shall replace the previous definition. The expression

LETTER (*opt_argument_list*)

shall invoke the function named *LETTER*. The behavior is undefined if the number of arguments in the invocation does not match the number of parameters in the definition. Functions shall be defined before they are invoked. A function shall be considered to be defined within its own body, so recursive calls shall be valid. The values of numeric constants within a function shall 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*)] shall cause termination of a function, popping of its auto variables, and specifies the result of the function. The first form shall be equivalent to `return(0)`. The value and scale of an invocation of the function shall be the value and scale of the expression in parentheses.

The `quit` statement (`quit`) shall 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 shall 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 shall be the value of the `scale` register when the function is invoked. The behavior is undefined if any of these functions is invoked with an argument outside the domain of the mathematical function.

4.3.8 Exit Status

The bc utility shall exit with one of the following values:

0	All input files were processed successfully.
unspecified	An error occurred.

4.3.9 Consequences of Errors

If any *file* operand is specified and the named file cannot be accessed, bc shall 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 noninteractive invocation of bc, invalid input causes undefined behavior.

4.3.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

This description is based on *BC—An Arbitrary Precision Desk-Calculator Language* by Lorinda Cherry and Robert Morris, in the BSD User Manual {B28}.

Automatic variables in bc do not work in exactly the same way as in either C or PL/1.

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)
}
```

History of Decisions Made

The bc utility is traditionally implemented as a front-end processor for dc; dc was not selected to be part of the standard because bc was thought to have a more intuitive programmatic interface. Current implementations that implement bc using dc are expected to be compliant.

The Exit Status for error conditions been left unspecified for several reasons:

(1) The bc utility is used in both interactive and noninteractive situations. Different exit codes may be appropriate for the two uses.

(2) It is unclear when a nonzero exit should be given; divide-by-zero, undefined functions, and syntax errors are all possibilities.

(3) It is not clear what utility the exit status has.

(4) In the 4.3 BSD, System V, and Ninth Edition implementations, bc works in conjunction with dc. dc is the parent, bc is the child. This was done to cleanly terminate bc if dc aborted.

The decision to have bc exit upon encountering an inaccessible input file is based on the belief that bc *file1 file2* is used most often when at least *file1* contains data/function declarations/initializations. Having bc continue with prerequisite files missing is probably not useful. There is no implication in the Consequences of Errors subclause that bc must check all its files for accessibility before opening any of them.

There was considerable debate on the appropriateness of the language accepted by bc. Several members of the balloting group preferred to see either a pure subset of the C language or some changes to make the language more compatible with C. While the `bc` language has some obvious similarities to C, it has never claimed to be compatible with any version of C. An interpreter for a subset of C might be a very worthwhile utility, and it could potentially make `bc` obsolete. However, no such utility is known in existing practice, and it was not within the scope of POSIX.2 to define such a language and utility. If and when they are defined, it may be appropriate to include them in a future revision of this standard. This left the following alternatives:

(1) Exclude any calculator language from the standard.

The consensus of the working group was that a simple programmatic calculator language is very useful. Also, an interactive version of such a calculator would be very important for the POSIX.2a revision. The only arguments for excluding any calculator were that it would become obsolete if and when a C-compatible one emerged, or that the absence would encourage the development of such a C-compatible one. These arguments did not sufficiently address the needs of current application writers.

(2) Standardize the existing `dc`, possibly with minor modifications.

The consensus of the working group was that `dc` is a fundamentally less usable language and that that would be far too severe a penalty for avoiding the issue of being similar to but incompatible with C.

(3) Standardize the existing `bc`, possibly with minor modifications.

This was the approach taken. Most of the proponents of changing the language would not have been satisfied until most or all of the incompatibilities with C were resolved. Since most of the changes considered most desirable would break existing applications and require significant modification to existing implementations, almost no modifications were made. The one significant modification that was made was the replacement of the traditional `bc`'s assignment operators `=+` et al. with the more modern `+=` et al. The older versions are considered to be fundamentally flawed because of the lexical ambiguity in uses like

```
a=-1
```

In order to permit implementations to deal with backward compatibility as they see fit, the behavior of this one ambiguous construct was made undefined. (At least three implementations have been known to support this change already, so the degree of change involved should not be great.)

The % operator is the mathematical remainder operator when `scale` is zero. The behavior of this operator for other values of `scale` is from traditional implementations of `bc`, and has been maintained for the sake of existing applications despite its nonintuitive nature.

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 period is also used in output.

Traditional implementations permit setting ibase and obase to a broader range of values. This includes values less than 2, which were not seen as sufficiently useful to standardize. These implementations do not interpret input properly for values of ibase outside greater than 16. This is because numeric constants are recognized syntactically, rather than lexically, as described in the standard. They are built from lexical tokens of single hexadecimal digits and periods. Since <blank>s between tokens are not visible at the syntactic level, it is not possible to properly recognize the multidigit ''digits'' used in the higher bases. The ability to recognize input in these bases was not considered useful enough to require modifying these implementations. Note that the recognition of numeric constants at the syntactic level is not a problem with conformance to the standard, as it does not impact the behavior of portable applications (and correct bc programs). Traditional implementations also accept input with all of the digits 0-9 and A-F regardless of the value of ibase; since digits with value greater than or equal to ibase are not really appropriate, the behavior when they appear is undefined, except for the common case of

```
ibase=8;
/* Process in octal base */
. . .
ibase=A
/* Restore decimal base */
```

In some historical implementations, if the expression to be written is an uninitialized array element, a leading <space> character and/or up to four leading 0 characters may be output before the character zero. This behavior is considered a bug; it is unlikely that any currently portable application relies on

```
echo 'b[3]' | bc
```

returning 00000 rather than 0.

Exact calculation of the number of fractional digits to output for a given value in a base other than 10 can be computationally expensive. Traditional implementations use a faster approximation, and this is permitted. Note that the requirements apply only to values of obase that the standard requires implementations to support (in particular, not to 1, 0, or negative bases, if an implementation supports them as an extension).

4.4 `cat` — Concatenate and print files

4.4.1 Synopsis

`cat [−u] [`*file* ...`]`

4.4.2 Description

The `cat` utility reads files in sequence and writes their contents to the standard output in the same sequence.

4.4.3 Options

The `cat` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−u` Write bytes from the input file to the standard output without delay as each is read.

4.4.4 Operands

The following operand shall be supported by the implementation:

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 shall read from the standard input at that point in the sequence. The `cat` utility shall not close and reopen standard input when it is referenced in this way, but shall accept multiple occurrences of − as a *file* operand.

4.4.5 External Influences

4.4.5.1 Standard Input

The standard input is used only if no *file* operands are specified, or if a *file* operand is −. See Input Files.

4.4.5.2 Input Files

The input files can be any file type.

4.4.5.3 Environment Variables

The following environment variables shall affect the execution of `cat`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.4.5.4 Asynchronous Events

Default.

4.4.6 External Effects

4.4.6.1 Standard Output

The standard output shall contain the sequence of bytes read from the input file(s). Nothing else shall be written to the standard output.

4.4.6.2 Standard Error

Used only for diagnostic messages.

4.4.6.3 Output Files

None.

4.4.7 Extended Description

None.

4.4.8 Exit Status

The cat utility shall exit with one of the following values:

- 0 All input files were output successfully.
- >0 An error occurred.

4.4.9 Consequences of Errors

Default.

4.4.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Historical versions of the cat utility include the options −e, −t, and −v, which permit the ends of lines, <tab>s, and invisible characters, respectively, to be rendered visible in the output. The working group omitted these options because they provide too fine a degree of control over what is made visible, and similar output can be obtained using a command such as:

```
sed -n -e 's/$/$/' -e l pathname
```

The −s option was omitted because it corresponds to different functions in BSD and System V-based systems. The BSD −s option to squeeze blank lines will be handled by more -s in the UPE. The System V −s option to silence error messages can be accomplished by redirecting the standard error. An alternative to cat −s is the following shell script using sed:

```
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
'
```

Note that the BSD documentation for `cat` uses the term ''blank line'' to mean the same as the POSIX ''empty line''; a line consisting only of a `<newline>`.

The BSD `−n` option is omitted because similar functionality can be obtained from the `−n` option of the `pr` utility.

The `−u` option is included here for its value in prototyping nonblocking reads from FIFO s. 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 BSD *setbuffer* () function, the System V *setbuf* () function, or the C Standard {7}*setvbuf* () function.

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.

Due to changes made to subclause 2.11.4 in Draft 11, the description of the *file* operand now states that – must be accepted multiple times, as in historical practice. This allows the command:

```
cat start - middle - end > file
```

when standard input is a terminal, to get 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.

History of Decisions Made

None.

4.5 `cd` — Change working directory

4.5.1 Synopsis

`cd` [*directory*]

4.5.2 Description

The `cd` utility shall change the working directory of the current shell execution environment; see 3.12.

When invoked with no operands, and the **HOME** environment variable is set to a nonempty value, the directory named in the **HOME** environment variable shall become the new working directory. If **HOME** is empty or is undefined, the default behavior is implementation defined.

4.5.3 Options

None.

4.5.4 Operands

The following operands shall be supported by the implementation:

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. If *directory* is –, the results are implementation defined.

4.5.5 External Influences

4.5.5.1 Standard Input

None.

4.5.5.2 Input Files

None.

4.5.5.3 Environment Variables

The following environment variables shall 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` shall search for *directory* relative to each directory named in the **CDPATH** variable, in the order listed. The new working directory shall 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 shall be treated as if it were an empty string.

HOME The name of the home directory, used when no *directory* operand is specified.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.5.5.4 Asynchronous Events

Default.

4.5.6 External Effects

4.5.6.1 Standard Output

If a nonempty directory name from **CDPATH** is used, an absolute pathname of the new working directory shall be written to the standard output as follows:

`"%s\n"`, *<new directory>*

Otherwise, there shall be no output.

4.5.6.2 Standard Error

Used only for diagnostic messages.

4.5.6.3 Output Files

None.

4.5.7 Extended Description

None.

4.5.8 Exit Status

The `cd` utility shall exit with one of the following values:

0 The directory was successfully changed.

>0 An error occurred.

4.5.9 Consequences of Errors

The working directory remains unchanged.

4.5.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Editor's Note: A balloter requested that the following rationale be highlighted in 2
the D11.2 recirculation. 2

Since `cd` 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 1
environment, such as one of the following: 1

```
(cd /tmp)                1
nohup cd                 1
find . -exec cd {} \;    1
```

it will not affect the working directory of the caller's environment. 1

The use of the **CDPATH** was introduced in the System V shell. Its use is analogous to the use of the **PATH** variable in the shell. Earlier systems such as the BSD C-shell used a shell parameter `cdpath` for this purpose.

History of Decisions Made

A common extension when **HOME** is undefined is to get the login directory from the user database for the invoking user. This does not occur on System V implementations.

Not included in this description are the features from the KornShell such as setting **OLDPWD**, toggling current and previous directory (`cd –`), and the two-operand form of `cd` (`cd` *old new*). This standard does not specify the results of `cd -` or of calls with more than one operand. Since these extensions are mostly used in interactive situations, they may be considered for inclusion in POSIX.2a. The result of `cd -` and of using no arguments with **HOME** unset or null have been made implementation defined at the request of the POSIX.6 security working group.

The setting of the **PWD** variable was removed from earlier drafts, as it can be replaced by `$(pwd)`.

4.6 chgrp — Change file group ownership

4.6.1 Synopsis

`chgrp [−R]` *group file* . . .

4.6.2 Description

The chgrp utility shall 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 shall perform actions equivalent to the POSIX.1 {8} *chown*() function, called with the following arguments:

(1) The *file* operand shall be used as the *path* argument.

(2) The user ID of the file shall be used as the *owner* argument.

(3) The specified *group ID* shall be used as the *group* argument.

4.6.3 Options

The chgrp utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

−R Recursively change file group IDs. For each *file* operand that names a directory, chgrp shall change the group of the directory and all files in the file hierarchy below it.

4.6.4 Operands

The following operands shall be supported by the implementation:

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.

4.6.5 External Influences

4.6.5.1 Standard Input

None.

4.6.5.2 Input Files

None.

4.6.5.3 Environment Variables

The following environment variables shall affect the execution of `chgrp`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.6.5.4 Asynchronous Events

Default.

4.6.6 External Effects

4.6.6.1 Standard Output

None.

4.6.6.2 Standard Error

Used only for diagnostic messages.

4.6.6.3 Output Files

None.

4.6.7 Extended Description

None.

4.6.8 Exit Status

The `chgrp` utility shall exit with one of the following values:

0	The utility executed successfully and all requested changes were made.
>0	An error occurred.

4.6.9 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 shall continue to process the remaining files in the hierarchy. If `chgrp` cannot read or search a directory within a hierarchy, it shall continue to process the other parts of the hierarchy that are accessible.

4.6.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status value. The working group chose to mask these by specifying only 0 and >0 as exit values.

History of Decisions Made

The functionality of `chgrp` is described substantially through references to functions in POSIX.1 {8}. In this way, there is no duplication of effort required for describing the interactions of permissions, multiple groups, etc.

4.7 **chmod** — Change file modes

4.7.1 Synopsis

`chmod [−R]` *mode file* . . .

4.7.2 Description

The `chmod` utility shall 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 defined whether and how the `chmod` utility affects any alternate or additional file access control mechanism (see *file access permissions* in 2.2.2.55) 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, shall be permitted to change the file mode bits of a file.

4.7.3 Options

The `chmod` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−R` Recursively change file mode bits. For each *file* operand that names a directory, `chmod` shall change the file mode bits of the directory and all files in the file hierarchy below it.

4.7.4 Operands

The following operands shall be supported by the implementation:

mode Represents the change to be made to the file mode bits of each file named by one of the *file* operands, as described in 4.7.7.

file A pathname of a file whose file mode bits are to be modified.

4.7.5 External Influences

4.7.5.1 Standard Input

None.

4.7.5.2 Input Files

None.

4.7.5.3 Environment Variables

The following environment variables shall affect the execution of `chmod`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.7.5.4 Asynchronous Events

Default.

4.7.6 External Effects

4.7.6.1 Standard Output

None.

4.7.6.2 Standard Error

Used only for diagnostic messages.

4.7.6.3 Output Files

None.

4.7.7 Extended Description

The *mode* operand shall be either a `symbolic_mode` expression or a nonnegative octal integer. The `symbolic_mode` form is described by the grammar in 4.7.7.1.

Each `clause` shall specify an operation to be performed on the current file mode bits of each *file*. The operations shall be performed on each *file* in the order in which the `clause`s are specified.

The *who* symbols `u`, `g`, and `o` shall specify the *user*, *group*, and *other* parts of the file mode bits, respectively. A *who* consisting of the symbol `a` shall 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` shall 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` shall 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 shall 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` shall represent the current permissions associated with the user, group, and other parts of the file mode bits, respectively. For the remainder of subclause 4.7.7 up to subclause 4.7.7.1, `perm` refers to the non-terminals `perm` and `permcopy` in the grammar in 4.7.7.1.

If multiple `actionlist`s are grouped with a single `wholist` in the grammar, each `actionlist` shall be applied in the order specified with that `wholist`. The `op` symbols shall represent the operation performed, as follows:

`+` If `perm` is not specified, the `+` operation shall 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, shall be set.

Otherwise, the file mode bits represented by the specified `who` and `perm` values shall be set.

`-` If `perm` is not specified, the `-` operation shall 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, shall be cleared.

Otherwise, the file mode bits represented by the specified `who` and `perm` values shall 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 standard.

If `perm` is not specified, the = operation shall 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, shall be set.

Otherwise, the file mode bits represented by the specified `who` and `perm` values shall be set.

When using the symbolic mode form on a regular file, it is implementation defined whether or not:

(1) 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,

(2) Requests to clear all execute bits also clear the set-user-ID-on-execution and set-group-ID-on-execution bits, or

(3) 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` (see 4.39.6.1) writes an `s` in the positions indicating that the set-user-ID-on-execution or set-group-ID-on-execution, the commands `chmod u-s file` or `chmod g-s file`, respectively, shall not be ignored.

When using the symbolic mode form on other file types, it is implementation 2
defined whether or not requests to set or clear the set-user-ID-on-execution or 2
set-group-ID-on-execution bits are honored. 2

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 shall not be modified. It shall not be an error to specify the `who` symbol `o` in conjunction with the `perm` symbol `s`.

For an octal integer *mode* operand, the file mode bits shall be set absolutely. The octal number form of the *mode* operand is obsolescent.

For each bit set in the octal number, the corresponding file permission bit shown 2
in the following table shall be set; all other file permission bits shall be cleared. 2
For regular files, for each bit set in the octal number corresponding to the set- 2
user-ID-on-execution or the set-group-ID-on-execution bits shown in the following 2
table shall be set; if these bits are not set in the octal number, they shall be 2
cleared. For other file types, it is implementation defined whether or not requests 2
to set or clear the set-user-ID-on-execution or set-group-ID-on-execution bits are 2
honored. 2

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 behavior is unspecified.

4.7.7.1 chmod Grammar

The grammar and lexical conventions in this subclause describe the syntax for the `symbolic_mode` operand. The general conventions for this style of grammar are described in 2.1.2. A valid `symbolic_mode` can be represented as the nonterminal symbol `symbolic_mode` in the grammar. Any discrepancies found between this grammar and descriptions in the rest of this clause shall be resolved in favor of this grammar.

The lexical processing shall be based entirely on single characters. Implementations need not allow `<blank>`s within the single argument being processed.

```
%start          symbolic_mode
%%

symbolic_mode : clause
              | symbolic_mode ',' clause
              ;

clause        : 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'
                    ;
```

4.7.8 Exit Status

The chmod utility shall exit with one of the following values:

- 0 The utility executed successfully and all requested changes were made.
- >0 An error occurred.

4.7.9 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 shall 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 shall continue to process the other parts of the hierarchy that are accessible.

4.7.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The functionality of chmod is described substantially through references to concepts defined in POSIX.1 {8}. In this way, there is less duplication of effort required for describing the interactions of permissions, etc. However, the behavior of this utility is not described in terms of the *chmod*() function from POSIX.1 {8}, because that specification requires certain side effects upon alternate file access control mechanisms that might not be appropriate, depending on the implementation.

Some historical 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. Since neither method is clearly better and users do not frequently try to make a hierarchy inaccessible to themselves, the standard does not specify what happens in this case.

Note that although the association shown in the table between bits in the octal number and the indicated file mode bits must be supported, this does not require that a conforming implementation has to actually use those octal values to implement the macros shown.

Historical System V implementations of `chmod` never use the process's *umask* when changing modes. Version 7 and historical BSD systems do use the mask when `who` is not specified, as described in this standard. Applications should 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*. Note that *mode* operands `−r`, `−w`, `−s`, `−x`, or `−X`, or anything beginning with a hyphen, must be preceded by `--` to keep it from being interpreted as an option.

It is difficult to express the grammar used by `chmod` in English, but the following examples have been accepted by historical System V and BSD systems and are, therefore, required to behave this way by POSIX.2 even though some of them could be expressed more succinctly:

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.

History of Decisions Made

Implementations that support mandatory file and record locking as specified by the */usr/group Standard* {B29} historically used the combination of set-group-ID bit set and group execute bit clear to indicate mandatory locking. This condition is usually set or cleared with the symbolic mode `perm` symbol `l` instead of the `perm` symbols `s` and `x` so that mandatory locking mode is not changed without explicit indication that that was what the user intended. Therefore, the details on how the implementation treats these conditions must be defined in the documentation. This standard does not require mandatory locking (nor does POSIX.1 {8}), but does allow it as an extension. However, POSIX.2 does require that the `ls` and `chmod` utilities work consistently in this area. If `ls -l file` says the set-group-ID bit is set, `chmod g-s file` must clear it (assuming appropriate privileges exist to change modes).

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status values. This problem is avoided here by specifying only 0 and >0 as exit values.

A ‘‘sticky’’ file mode bit, indicating that the text portion of an executable object program file should be saved after the program is gone, has meaning in some implementations, but was omitted here because its purpose is implementation dependent and because it was omitted from POSIX.1 {8}. On 4.3BSD-based implementations, the sticky bit is used in conjunction with directory permissions to keep anyone from deleting a file that they do not own from the directory. The `perm` symbol `t` is used to represent the sticky bit in many existing implementations and should not be used for other conflicting extensions.

POSIX.1 {8} indicates that implementation-defined restrictions may cause the S_ISUID and S_ISGID bits to be ignored. POSIX.2 allows the `chmod` utility to choose to modify these bits before calling POSIX.1 {8} *chmod*() (or some function providing equivalent capabilities) for nonregular files. Among other things, this allows implementations that use the set-user-ID and set-group-ID bits on directories to enable extended features to handle these extensions in an intelligent manner. Portable applications should never assume that they know how these bits will be interpreted, except on regular files.

The grammar in Draft 9 did not allow several symbolic mode operands that are correctly processed by historical implementations. (It only allowed two clauses and one `op` per clause.) The grammar presented in Draft 10 matches historical implementations.

The `X` `perm` symbol was added, as provided in BSD-based systems, because it provides commonly desired functionality when doing recursive (–R option) modifications. Similar functionality is not provided by the `find` utility. Historical BSD versions of `chmod`, however, only supported `X` with `op +`; it has been extended here because it is also useful with `op =`. (It has also been added for `op -` even though it duplicates `x`, in this case, because it is intuitive and easier to explain.)

The grammar was extended with the `permcopy` nonterminal to allow existing-practice forms of symbolic modes like `o=u-g` (i.e., set the ‘‘other’’ permissions to the permissions of ‘‘owner’’ minus the permissions of ‘‘group’’.)

4.8 chown — Change file ownership

4.8.1 Synopsis

`chown [−R]` *owner* [**:** *group*] *file* . . .

4.8.2 Description

The `chown` utility shall 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 shall perform actions equivalent to the POSIX.1 {8} *chown*() function, called with the following arguments:

(1) The *file* operand shall be used as the *path* argument.

(2) The user ID indicated by the *owner* portion of the first operand shall be used as the *owner* argument.

(3) If the *group* portion of the first operand is given, the group ID indicated by it shall be used as the *group* argument; otherwise, the group ID of the file shall be used as the *group* argument.

4.8.3 Options

The `chown` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−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.

4.8.4 Operands

The following operands shall be supported by the implementation:

owner [**:** *group*]
A user ID and optional group ID to be assigned to file. The *owner* portion of this operand shall 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 is used as the user ID. Similarly, if the *group* portion of this operand is present, it shall 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 shall be used as the group ID.

file A pathname of a file whose user ID is to be modified.

4.8.5 External Influences

4.8.5.1 Standard Input

None.

4.8.5.2 Input Files

None.

4.8.5.3 Environment Variables

The following environment variables shall affect the execution of `chown`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.8.5.4 Asynchronous Events

Default.

4.8.6 External Effects

4.8.6.1 Standard Output

None.

4.8.6.2 Standard Error

Used only for diagnostic messages.

4.8.6.3 Output Files

None.

4.8.7 Extended Description

None.

4.8.8 Exit Status

The `chown` utility shall exit with one of the following values:

0	The utility executed successfully and all requested changes were made.
>0	An error occurred.

4.8.9 Consequences of Errors

If, when invoked with the `−R` option, `chown` attempts but fails to change the user ID and/or, if the *group* operand is specified, group ID, of a particular file in a specified file hierarchy, it shall continue to process the remaining files in the hierarchy.

If `chown` cannot read or search a directory within a hierarchy, it shall continue to process the other parts of the hierarchy that are accessible.

4.8.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The System V and BSD versions use different exit status codes. Some implementations used the exit status as a count of the number of errors that occurred; this practice is unworkable since it can overflow the range of valid exit status values. These are masked by specifying only 0 and >0 as exit values.

The functionality of `chown` is described substantially through references to functions in POSIX.1 {8}. In this way, there is no duplication of effort required for describing the interactions of permissions, multiple groups, etc.

For implementations on which symbolic links are supported, actual use of the *chown*() function to implement this utility might not be the appropriate, depending on the implementation.

History of Decisions Made

The 4.3BSD method of specifying both owner and group was included in this standard because:

(1) There are cases where the desired end condition could not be achieved using the `chgrp` and `chown` (that only changed the user ID) utilities. [If the current owner is not a member of the desired group and the desired owner is not a member of the current group, the *chown*() function could fail unless both owner and group are changed at the same time.]

(2) Even if they could be changed independently, in cases where both are being changed, there is a 100 percent performance penalty caused by being forced to invoke both utilities.

The BSD syntax *user*[.*group*] was changed to *user*[:*group*] in POSIX.2 because the period is a valid character in login names (as specified by POSIX.1 {8}, login names consist of characters in the portable filename character set). The colon character was chosen as the replacement for the period character because it would never be allowed as a character in a user name or group name on traditional implementations.

The `−R` option is considered by some observers as an undesirable departure from the traditional UNIX system tools approach; since a tool, `find`, already exists to recurse over directories, there was felt to be no good reason to require other tools to have to duplicate that functionality. However, the `−R` option was deemed an important user convenience, is far more efficient than forking a separate process for each element of the directory hierarchy, and is in widespread historical use.

4.9 `cksum` — Write file checksums and sizes

4.9.1 Synopsis

`cksum` [*file* . . .]

4.9.2 Description

The `cksum` utility shall calculate and write to standard output a cyclic redundancy check (CRC) for each input file, and also write 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 networking standard ISO 8802-3 {B7}.

The CRC checksum shall be obtained in the following way:

The encoding 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 shall be 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 shall be used.

(2) $M(x)$ is multiplied by x^{32} (i.e., 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.

4.9.3 Options

None.

4.9.4 Operands

The following operand shall be supported by the implementation:

file A pathname of a file to be checked. If no *file* operands are specified, the standard input is used.

4.9.5 External Influences

4.9.5.1 Standard Input

The standard input is used only if no *file* operands are specified. See Input Files.

4.9.5.2 Input Files

The input files can be any file type.

4.9.5.3 Environment Variables

The following environment variables shall affect the execution of `cksum`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.9.5.4 Asynchronous Events

Default.

4.9.6 External Effects

4.9.6.1 Standard Output

For each file processed successfully, the `cksum` utility shall write in the following format: 2

`"%u %d %s\n"`, <*checksum*> , <*# of octets*>, <*pathname*> 2

If no *file* operand was specified, the pathname and its leading space shall be omitted.

4.9.6.2 Standard Error

Used only for diagnostic messages.

4.9.6.3 Output Files

None.

4.9.7 Extended Description

None.

4.9.8 Exit Status

The `cksum` utility shall exit with one of the following values:

- 0 All files were processed successfully.
- >0 An error occurred.

4.9.9 Consequences of Errors

Default.

4.9.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `cksum` utility is typically used to quickly compare a suspect file against a trusted version of the same. However, no claims are made by POSIX.2 that this comparison is cryptographically secure; the historical `sum` utility from which `cksum` was inspired has traditionally been used mainly to ensure that files transmitted over noisy media arrive intact. 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 POSIX.1 {8}. Since POSIX.2 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 2
transmitted between two systems and undergoes any data transformation (such 2
as moving 8-bit characters into 9-bit bytes or changing ''little Endian'' byte order- 2
ing to ''big Endian''), identical CRC values cannot be expected. Implementations 2
performing such transformations may extend `cksum` to handle such situations. 2

The following C-language program can be used as a model to describe the algorithm. It assumes that a `char` is one octet. It also assumes that the entire file is 2 available for one pass through the function. This was done for simplicity in demonstrating the algorithm, rather than as an implementation model. 2

```
static unsigned long crctab[] = {                                    2
0x0,                                                                 2
0x77073096, 0xee0e612c, 0x990951ba, 0x076dc419, 0x706af48f,
0xe963a535, 0x9e6495a3, 0x0edb8832, 0x79dcb8a4, 0xe0d5e91e,
0x97d2d988, 0x09b64c2b, 0x7eb17cbd, 0xe7b82d07, 0x90bf1d91,
0x1db71064, 0x6ab020f2, 0xf3b97148, 0x84be41de, 0x1adad47d,
0x6ddde4eb, 0xf4d4b551, 0x83d385c7, 0x136c9856, 0x646ba8c0,
0xfd62f97a, 0x8a65c9ec, 0x14015c4f, 0x63066cd9, 0xfa0f3d63,
0x8d080df5, 0x3b6e20c8, 0x4c69105e, 0xd56041e4, 0xa2677172,
0x3c03e4d1, 0x4b04d447, 0xd20d85fd, 0xa50ab56b, 0x35b5a8fa,
0x42b2986c, 0xdbbbc9d6, 0xacbcf940, 0x32d86ce3, 0x45df5c75,
0xdcd60dcf, 0xabd13d59, 0x26d930ac, 0x51de003a, 0xc8d75180,
0xbfd06116, 0x21b4f4b5, 0x56b3c423, 0xcfba9599, 0xb8bda50f,
0x2802b89e, 0x5f058808, 0xc60cd9b2, 0xb10be924, 0x2f6f7c87,
0x58684c11, 0xc1611dab, 0xb6662d3d, 0x76dc4190, 0x01db7106,
0x98d220bc, 0xefd5102a, 0x71b18589, 0x06b6b51f, 0x9fbfe4a5,
0xe8b8d433, 0x7807c9a2, 0x0f00f934, 0x9609a88e, 0xe10e9818,
0x7f6a0dbb, 0x086d3d2d, 0x91646c97, 0xe6635c01, 0x6b6b51f4,
0x1c6c6162, 0x856530d8, 0xf262004e, 0x6c0695ed, 0x1b01a57b,
0x8208f4c1, 0xf50fc457, 0x65b0d9c6, 0x12b7e950, 0x8bbeb8ea,
0xfcb9887c, 0x62dd1ddf, 0x15da2d49, 0x8cd37cf3, 0xfbd44c65,
0x4db26158, 0x3ab551ce, 0xa3bc0074, 0xd4bb30e2, 0x4adfa541,
0x3dd895d7, 0xa4d1c46d, 0xd3d6f4fb, 0x4369e96a, 0x346ed9fc,
0xad678846, 0xda60b8d0, 0x44042d73, 0x33031de5, 0xaa0a4c5f,
0xdd0d7cc9, 0x5005713c, 0x270241aa, 0xbe0b1010, 0xc90c2086,
0x5768b525, 0x206f85b3, 0xb966d409, 0xce61e49f, 0x5edef90e,
0x29d9c998, 0xb0d09822, 0xc7d7a8b4, 0x59b33d17, 0x2eb40d81,
0xb7bd5c3b, 0xc0ba6cad, 0xedb88320, 0x9abfb3b6, 0x03b6e20c,
0x74b1d29a, 0xead54739, 0x9dd277af, 0x04db2615, 0x73dc1683,
0xe3630b12, 0x94643b84, 0x0d6d6a3e, 0x7a6a5aa8, 0xe40ecf0b,
0x9309ff9d, 0x0a00ae27, 0x7d079eb1, 0xf00f9344, 0x8708a3d2,
0x1e01f268, 0x6906c2fe, 0xf762575d, 0x806567cb, 0x196c3671,
0x6e6b06e7, 0xfed41b76, 0x89d32be0, 0x10da7a5a, 0x67dd4acc,
0xf9b9df6f, 0x8ebeeff9, 0x17b7be43, 0x60b08ed5, 0xd6d6a3e8,
0xa1d1937e, 0x38d8c2c4, 0x4fdff252, 0xd1bb67f1, 0xa6bc5767,
0x3fb506dd, 0x48b2364b, 0xd80d2bda, 0xaf0a1b4c, 0x36034af6,
0x41047a60, 0xdf60efc3, 0xa867df55, 0x316e8eef, 0x4669be79,
0xcb61b38c, 0xbc66831a, 0x256fd2a0, 0x5268e236, 0xcc0c7795,
0xbb0b4703, 0x220216b9, 0x5505262f, 0xc5ba3bbe, 0xb2bd0b28,
0x2bb45a92, 0x5cb36a04, 0xc2d7ffa7, 0xb5d0cf31, 0x2cd99e8b,
0x5bdeae1d, 0x9b64c2b0, 0xec63f226, 0x756aa39c, 0x026d930a,
0x9c0906a9, 0xeb0e363f, 0x72076785, 0x05005713, 0x95bf4a82,
0xe2b87a14, 0x7bb12bae, 0x0cb61b38, 0x92d28e9b, 0xe5d5be0d,
0x7cdcefb7, 0x0bdbdf21, 0x86d3d2d4, 0xf1d4e242, 0x68ddb3f8,
0x1fda836e, 0x81be16cd, 0xf6b9265b, 0x6fb077e1, 0x18b74777,
0x88085ae6, 0xff0f6a70, 0x66063bca, 0x11010b5c, 0x8f659eff,
0xf862ae69, 0x616bffd3, 0x166ccf45, 0xa00ae278, 0xd70dd2ee,
0x4e048354, 0x3903b3c2, 0xa7672661, 0xd06016f7, 0x4969474d,
0x3e6e77db, 0xaed16a4a, 0xd9d65adc, 0x40df0b66, 0x37d83bf0,
```

```
0xa9bcae53, 0xdebb9ec5, 0x47b2cf7f, 0x30b5ffe9, 0xbdbdf21c,
0xcabac28a, 0x53b39330, 0x24b4a3a6, 0xbad03605, 0xcdd70693,
0x54de5729, 0x23d967bf, 0xb3667a2e, 0xc4614ab8, 0x5d681b02,
0x2a6f2b94, 0xb40bbe37, 0xc30c8ea1, 0x5a05df1b, 0x2d02ef8d
};

unsigned long memcrc(const unsigned char *b, size_t n)
{
/*      Input arguments:
 *      const char*     b == byte sequence to checksum
 *      size_t          n == length of sequence
 */

        register unsigned int   i, c, s = 0;

        for (i = n; i > 0; --i) {
                c = (unsigned int)(*b++);
                s = (s << 8) ^ crctab[(s >> 24) ^ c];
        }

        /* extend with the length of the string */
        while (n != 0) {
                c = n & 0377;
                n >>= 8;
                s = (s << 8) ^ crctab[(s >> 24) ^ c];
        }

        return ~s;
}
```

History of Decisions Made

The historical practice of writing the number of ‘‘blocks’’ has been removed in favor of writing the number of octets since the latter is not only more useful, but historical implementations have not been consistent in defining what a ‘‘block’’ meant. Octets are used instead of bytes because bytes can differ in size between systems.

The algorithm used was selected to increase the robustness of the utility’s operation. Neither the System V nor BSD `sum` algorithm was selected. Since each of these was different and each was the default behavior on those systems, no realistic compromise was available if either were selected—some set of historical applications would break. Therefore, the name was changed to `cksum`. Although the historical `sum` commands will probably continue to be provided for many years to come, programs designed for portability across systems should use the new name.

The algorithm selected is based on that used by the Ethernet standard for the Frame Check Sequence Field. The algorithm used does not match the technical definition of a *checksum*; the term is used for historical reasons. The length of the file is included in the CRC calculation because this parallels Ethernet’s inclusion of a length field in its CRC, but also because it guards against inadvertent collisions between files that begin with different series of zero octets. The chance that two different files will produce identical CRCs is much greater when their lengths are not considered. Keeping the length and the checksum of the file itself separate would yield a slightly more robust algorithm, but historical usage has always been that a single number (the checksum as printed) represents the signature of the file. It was decided that historical usage was the more important consideration.

Earlier drafts contained modifications to the Ethernet algorithm that involved extracting table values whenever an intermediate result became zero. This was demonstrated to be less robust than the current method and mathematically difficult to describe or justify.

Editor's Note: The following bibliographic references will be cleaned up before the standard is completed.

The calculation used is identical to that given in pseudo-code on page 1011 of *Communications of the ACM*, August, 1988 in the article ''Computation of Cyclic Redundancy Checks Via Table Lookup'' by Dilip V. Sarwate. The pseudo-code rendition is:

```
      X <- 0;  Y <- 0;
      for i <- m -1 step -1 until 0 do
              begin
              T <- X(1) ^ A[i];
              X(1) <- X(0); X(0) <- Y(1); Y(1) <- Y(0); Y(0) <- 0;
              comment: f[T] and f'[T] denote the T-th words in the
                      table f and f' ;
              X <- X ^ f[T]; Y <- Y ^ f'[T];
              end
```

The pseudo-code is reproduced exactly as given; however, note that in `cksum`'s case, `A[i]` represents a byte of the file, the words `X` and `Y` are a treated as a single 32-bit value, and the tables `f` and `f'` are a single table containing 32-bit values.

The article also discusses generating the table(s).

Other sources consulted about CRC's:

> ''A Tutorial on CRC Computations,'' Ramabadran and Gaitonde, *IEEE Micro,* p. 62, August 1988;
>
> *Computer Networks*, Andrew Tanenbaum, Prentice-Hall, Inc.

4.10 **cmp** — Compare two files

4.10.1 Synopsis

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

4.10.2 Description

The `cmp` utility shall compare two files. The `cmp` utility shall write no output if the files are the same. Under default options, if they differ, it shall write to standard output the byte and line number at which the first difference occurred. Bytes and lines shall be numbered beginning with 1.

4.10.3 Options

The `cmp` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

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

`-s` Write nothing for differing files; return exit status only.

4.10.4 Operands

The following operands shall be supported by the implementation:

file1 A pathname of the first file to be compared. If *file1* is −, the standard input shall be used.

file2 A pathname of the second file to be compared. If *file2* is −, the standard input shall 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.

4.10.5 External Influences

4.10.5.1 Standard Input

The standard input shall be used only if the *file1* or *file2* operand refers to standard input. See Input Files.

4.10.5.2 Input Files

The input files can be any file type.

4.10.5.3 Environment Variables

The following environment variables shall affect the execution of `cmp`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.10.5.4 Asynchronous Events

Default.

4.10.6 External Effects

4.10.6.1 Standard Output

In the POSIX Locale, results of the comparison shall be written to standard output. When no options are used, the format shall 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>* shall be relative to the 2
beginning of the file, beginning with 1. 2

The *<additional info>* field shall either be null or a string that starts with a 1
`<blank>` and contains no `<newline>` characters. 1

No output shall be written to standard output when the `−s` option is used.

4.10.6.2 Standard Error

Used only for diagnostic messages. If *file1* and *file2* are identical for the entire 2
length of the shorter file, in the POSIX Locale the following diagnostic message 2
shall be written, unless the `−s` option is specified. 2

`"cmp: EOF on %s%s\n"`, *<name of shorter file>*, *<additional info>* 1

4.10.6.3 Output Files

None.

4.10.7 Extended Description

None.

4.10.8 Exit Status

The `cmp` utility shall exit with one of the following values:

- 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.

4.10.9 Consequences of Errors

Default.

4.10.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The global language in Section 2 indicates that using two mutually-exclusive options together produces unspecified results. Some System V implementations consider the option usage:

```
cmp −l −s ...
```

to be an error. They also treat:

```
cmp −s −l ...
```

as if no options were specified. Both of these behaviors are considered bugs, but are allowed.

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 POSIX.1 {8}. Since POSIX.2 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.

The word `char` in the standard output format comes from historical usage, even though it is actually a byte number. When `cmp` is supported in other locales, implementations are encouraged to use the word `byte` or its equivalent in another language. Users should not interpret this difference to indicate that the functionality of the utility changed between locales.

History of Decisions Made

Some systems report on the number of lines in the identical-but-shorter file case. This is allowed by the inclusion of the <*additional info*> fields in the output format. The restriction on having a leading `<blank>` and no `<newline>`s is to make parsing for the file name easier. It is recognized that some file names containing white-space characters will make parsing difficult anyway, but the restriction does aid programs used on systems where the names are predominantly well behaved.

4.11 comm — Select or reject lines common to two files

4.11.1 Synopsis

`comm [−123]` *file1 file2*

4.11.2 Description

The `comm` utility shall 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.

4.11.3 Options

The `comm` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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* . 1

4.11.4 Operands

The following operands shall be supported by the implementation:

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.

4.11.5 External Influences

4.11.5.1 Standard Input

The standard input shall be used only if one of the *file1* or *file2* operands refers to standard input. See Input Files.

4.11.5.2 Input Files

The input files shall be text files.

4.11.5.3 Environment Variables

The following environment variables shall affect the execution of `comm`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

LC_COLLATE This variable shall determine the locale for the collating sequence `comm` expects to have been used when the input files were sorted.

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

4.11.5.4 Asynchronous Events

Default.

4.11.6 External Effects

4.11.6.1 Standard Output

The `comm` utility shall produce output depending on the options selected. If the `−1`, `−2`, and `−3` options are all selected, `comm` shall write nothing to standard output.

If the `−1` option is not selected, lines contained only in *file1* shall be written using the format:

`"%s\n"`, *<line in file1>*

If the `−2` option is not selected, lines contained only in *file2* shall 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 shall 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 shall be in the collating sequence of the original lines.

4.11.6.2 Standard Error

Used only for diagnostic messages.

4.11.6.3 Output Files

None.

4.11.7 Extended Description

None.

4.11.8 Exit Status

The `comm` utility shall exit with one of the following values:

0	All input files were successfully output as specified.
>0	An error occurred.

4.11.9 Consequences of Errors

Default.

4.11.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

If the input files are not properly presorted, the output of `comm` might not be useful.

If a file named `posix.2` contains a sorted list of the utilities in this standard, a file named `xpg3` contains a sorted list of the utilities specified in 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 posix.2 xpg3 | comm -23 - svid89
```

would print a list of utilities in this standard not specified by either of the other documents,

```
comm -12 posix.2 xpg3 | comm -12 - svid89
```

would print a list of utilities specified by all three documents, and

```
comm -12 xpg3 svid89 | comm -23 - posix.2
```

would print a list of utilities specified by both XPG3 and *SVID*, but not specified in this standard.

History of Decisions Made

None.

4.12 `command` — Execute a simple command

4.12.1 Synopsis

`command [−p]` *command_name* [*argument* ...]

4.12.2 Description

The `command` utility shall cause the shell to treat the arguments as a simple com-
mand, suppressing the shell function lookup that is described in 3.9.1.1 item 1
(1)(b). 1

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 3.14 shall not occur. In every other respect, if *command_name* is not the name of a function, the effect of `command` shall be the same as omitting `command`.

4.12.3 Options

The `command` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−p` Perform the command search using a default value for **PATH** that is guaranteed to find all of the standard utilities.

4.12.4 Operands

The following operands shall be supported by the implementation:

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.

4.12.5 External Influences

4.12.5.1 Standard Input

None.

4.12.5.2 Input Files

None.

4.12.5.3 Environment Variables

The following environment variables shall affect the execution of `command`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters).

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

PATH This variable shall determine the search path used during the command search described in 3.9.1.1, except as described under the `−p` option.

4.12.5.4 Asynchronous Events

Default.

4.12.6 External Effects

4.12.6.1 Standard Output

None.

4.12.6.2 Standard Error

Used only for diagnostic messages.

4.12.6.3 Output Files

None.

4.12.7 Extended Description

None.

4.12.8 Exit Status

The `command` utility shall exit with one of the following values:

126 The utility specified by *command_name* was found but could not be 1
invoked. 1

127 An error occurred in the `command` utility or the utility specified by 1
command_name could not be found. 1

Otherwise, the exit status of `command` shall be that of the simple command specified by the arguments to `command`.

4.12.9 Consequences of Errors

Default.

4.12.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The order for command search in POSIX.2 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
```

Since `command` appears in Table 2-2, it will always be found prior to the **PATH** search.

There is nothing in the description of `command` that implies the command line is parsed any differently than for any other simple command. For example,

```
command a | b ; c
```

is not parsed in any special way that causes `|` or `;` to be treated other than a pipe operator or semicolon or that prevents function lookup on `b` or `c`.

Examples : Make a version of `cd` that always prints out the new working directory exactly once:

```
cd() {
        command cd "$@" >/dev/null
        pwd
}
```

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.                              1

\unset -f command
#       Ensure command is not a user function.
#       Note that unset is escaped to prevent an alias being used
#       for unset on implementations that support aliases.

PATH="$(\command -p getconf _CS_PATH):$PATH"
#       Put on a reliable PATH prefix.

#       Now, unset all utility names that will be used (or
#       invoke them with \command each time).
#       . . .
```

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 aliases and/or functions to exist when it is invoked; neither capability is specified by POSIX. 2, but neither is prohibited as an extension. For example, the proposed UPE supplement to POSIX. 2 introduces a **ENV** variable that precedes the invocation of the script with a user startup script. Such a script could have used the aliasing facility from the UPE or the functions in POSIX. 2 to spoof the application.

The `command`, `env`, `nohup`, and `xargs` utilities have been specified to use exit
code 127 if an error occurs so that applications can distinguish ''failure to find a 1
utility'' from ''invoked utility exited with an error indication.'' The value 127 was 1
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 con-
fused with termination due to receipt of a signal. The value 126 was chosen in a 1
similar manner to indicate that the utility could be found, but not invoked. Some 1
scripts produce meaningful error messages differentiating the 126 and 127 cases. 1
The distinction between exit codes 126 and 127 is based on KornShell practice 2
that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 2
126 when any attempt to *exec* the utility fails for any other reason. 2

History of Decisions Made

The `command` utility is somewhat similar to the Eighth Edition `builtin` command, but since `command` also goes to the file system to search for utilities, the name `builtin` would not be intuitive.

The `command` utility will most likely be provided as a regular built-in. In an earlier draft, it was a special built-in. This was changed for the following reasons:

— The removal of exportable functions made the special precedence of a special built-in unnecessary.

— A special built-in has special properties (see the enumerated list at the beginning of 3.14) that were inappropriate for invoking other utilities. For example, two commands such as

`date >` *unwritable-file*

`command date >` *unwritable-file*

would have entirely different results; in a noninteractive script, the former would continue to execute the next command, the latter would abort. Introducing this semantic difference along with suppressing functions was seen to be nonintuitive.

— There are some advantages of suppressing the special characteristics of special built-ins on occasion. For example:

`command exec >` *unwritable-file*

will not cause a noninteractive script to abort, so that the output status can be checked by the script.

An earlier draft presented a larger number of options. Most were removed because they were not useful to real portable applications, given the new command search order.

The `−p` option is present because it is useful to be able to ensure a safe path search that will find all the POSIX.2 standard utilities. This search might not be identical to the one that occurs through one of the POSIX.1 {8} *exec* functions when **PATH** is unset, as explained in 2.6.1. At the very least, this feature is required to allow the script to access the correct version of `getconf` so that the value of the default path can be accurately retrieved.

4.13 cp — Copy files

4.13.1 Synopsis

`cp [−fip]` *source_file target_file* 2

`cp [−fip]` *source_file ... target* 2

`cp −R [−fip]` *source_file ... target* 2

`cp −r [−fip]` *source_file ... target* 2

4.13.2 Description

The first synopsis form is denoted by two operands, neither of which are existing files of type directory. The `cp` utility shall 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 shall be 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 POSIX.1 {8}, but is not a file of type directory. The `cp` utility shall 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 shall 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 shall 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* shall be *target;* the name of the corresponding destination path for all other files in the file hierarchy shall be the concatenation of *target,* a slash character, and the pathname of the file relative to *source_file.*

It shall be 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 POSIX.1 {8}, 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 shall be taken:

(1) If *source_file* references the same file as *dest_file,* `cp` may write a diagnostic message to standard error; it shall do nothing more with *source_file* 1
and shall go on to any remaining files. 1

(2) If *source_file* is of type directory, the following steps shall be taken:

(a) If neither the `−R` or `−r` options were specified, `cp` shall 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` shall 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 POSIX.1 {8}, the behavior is implementation defined.

(d) If *dest_file* exists and it is not of type directory, `cp` shall 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 shall 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` shall 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` shall attempt to copy files in the file hierarchy rooted in *source_file.*

(f) The files in the directory *source_file* shall 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 shall 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 shall 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 shall be taken: 1

(a) If *dest_file* exists, the following steps are taken:

[1] If the `−i` option is in effect, the `cp` utility shall write a prompt to the standard error and read a line from the standard input. If the response is not affirmative, `cp` shall do nothing more with *source_file* and go on to any remaining files.

[2] A file descriptor for *dest_file* shall be obtained by performing actions equivalent to the POSIX.1 {8} *open*() function call using *dest_file* as the *path* argument, and the bitwise inclusive OR of O_WRONLY and O_TRUNC as the *oflag* argument.

[3] If the attempt to obtain a file descriptor fails and the `−f` option is in effect, `cp` shall attempt to remove the file by performing actions equivalent to the POSIX.1 {8} *unlink*() function called using *dest_file* as the *path* argument. If this attempt succeeds, `cp` shall continue with step (3b).

(b) If *dest_file* does not exist, a file descriptor shall be obtained by performing actions equivalent to the POSIX.1 {8} *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* shall be the *mode* argument.

(c) If the attempt to obtain a file descriptor fails, `cp` shall 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* shall be written to the file descriptor. Any write errors shall cause `cp` to write a diagnostic message to standard error and continue to step (3)(e).

(e) The file descriptor shall be closed.

(f) The `cp` utility shall 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` shall go on to any remaining files.

(4) Otherwise, the following steps shall be taken:

(a) If the `−r` option was specified, the behavior is implementation defined.

(b) If the `−R` option was specified, the following steps shall be taken:

[1] The *dest_file* shall be created with the same file type as *source_file* .

[2] If *source_file* is a file of type FIFO, the file permission bits shall 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 defined.

If this creation fails for any reason, `cp` shall 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 2.2.2.55), their effect on copies of files is implementation-defined.

4.13.3 Options

The `cp` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−f` If a file descriptor for a destination file cannot be obtained, as 2
described in step (3a)[2], attempt to unlink the destination file 2
and proceed. 2

`−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 shall 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` shall 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-defined, bits may be duplicated as well. If this duplication fails for any reason, `cp` shall 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 shall 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* shall not be deleted if these characteristics cannot be preserved.

`−R` Copy file hierarchies.

`−r` Copy file hierarchies. The treatment of special files is implemen- 1
tation defined. 1

4.13.4 Operands

The following operands shall be supported by the implementation:

source_file A pathname of a file to be copied.

target_file A pathname of an existing or nonexisting file, used for the output when a single file is copied.

target A pathname of a directory to contain the copied file(s).

4.13.5 External Influences

4.13.5.1 Standard Input

Used to read an input line in response to each prompt specified in Standard Error. Otherwise, the standard input shall not be used.

4.13.5.2 Input Files

The input files specified as operands may be of any file type.

4.13.5.3 Environment Variables

The following environment variables shall affect the execution of `cp`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_MESSAGES This variable shall determine the processing of affirmative responses and the language in which messages should be written.

4.13.5.4 Asynchronous Events

Default.

4.13.6 External Effects

4.13.6.1 Standard Output

None.

4.13.6.2 Standard Error

A prompt shall be written to standard error under the conditions specified in 4.13.2. The prompt shall contain the destination pathname, but its format is otherwise unspecified. Otherwise, the standard error shall be used only for diagnostic messages.

4.13.6.3 Output Files

The output files may be of any type.

4.13.7 Extended Description

None.

4.13.8 Exit Status

The `cp` utility shall exit with one of the following values:

0 No error occurred.

>0 An error occurred.

4.13.9 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.

4.13.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

2

The −i option exists on BSD systems, giving applications and users a way to avoid accidentally removing files when copying. Although the 4.3BSD version does not prompt if the standard input is not a terminal, the working group decided that use of −i is a request for interaction, so when the destination path exists, the utility takes instructions from whatever responds on standard input.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified, because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application using the −i option relies on the system to provide the most suitable dialogue directly with the user, based on the behavior specified.

The −p option is historical practice on BSD systems, duplicating the time of last data modification and time of last access. POSIX.2 extends it to preserve the user and group IDs, as well as the file permissions. This requirement has obvious problems in that the directories are almost certainly modified after being copied. This specification requires that the modification times be preserved even so. The statement that the order in which the characteristics are duplicated is unspecified is to permit implementations to provide the maximum amount of security for the user. Implementations should take into account the obvious security issues involved in setting the owner, group, and mode in the wrong order or creating files with an owner, group, or mode different from the final value.

It is unspecified whether cp writes diagnostic messages when the user and group IDs cannot be set due to the widespread practice of users using −p to duplicate some portion of the file characteristics, indifferent to the duplication of others. Historic implementations only write diagnostic messages on errors other than [EPERM].

The −r option is historical practice on BSD and BSD-derived systems, copying file hierarchies as opposed to single files. This functionality is used heavily in existing applications and its loss would significantly decrease consensus. The −R option was added as a close synonym to the −r option, selected for consistency with all other options in the standard that do recursive directory descent.

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 than 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 current cp utility specification is intended to require that the −R option recreate the file hierarchy and that the −r option support historical practice. It is anticipated that a future

version of this standard will deprecate the `−r` option, and for that reason, there has been no attempt to fix its behavior with respect to FIFOs or other file types
where copying the file is clearly wrong. However, some systems support `−r` with 1
the same abilities as the `−R` defined in POSIX.2. To accommodate them as well as 1
systems that do not, the differences between `−r` and `−R` are implementation 1
defined. Implementations may make them identical. 1

When a failure occurs during the copying of a file hierarchy, `cp` is required to attempt to copy files that are on the same level in the hierarchy or above the file where the failure occurred. It is unspecified if `cp` shall attempt to copy files below the file where the failure occurred (which cannot succeed in any case).

Permissions, owners, and groups of created special file types have been deliberately left as implementation defined. This is to allow systems to satisfy special requirements (for example, allowing users to create character special devices, but requiring them to be owned by a certain group). In general, it is strongly suggested that the permissions, owner, and group be the same as if the user had run the traditional `mknod`, `ln`, or other utility to create the file. It is also probable that additional privileges will be required to create block, character, or other, implementation-specific, special file types.

Additionally, the `−p` option explicitly requires that all set-user-ID and set-group- 1
ID permissions be discarded if any of the owner or group IDs cannot be set. This is to keep users from unintentionally giving away special privilege when copying programs.

When creating regular files, historical versions of `cp` use the mode of the source file as modified by the file mode creation mask. Other choices would have been to use the mode of the source file unmodified by the creation mask, or to use the same mode as would be given to a new file created by the user, plus the execution bits of the source file, and then modified by the file mode creation mask. In the absence of any strong reason to change historic practice, it was in large part retained.

The one difference is that 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/set-group-ID to them when copying files or to make set-user-ID/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 have execute permission to a set-user-ID program than did previously. In particular, this is a problem for super-users copying users' trees. A finer granularity of protection could be specified, in that the set-user-ID/set-group-ID bits could be retained under certain conditions even if the owner or group could not be set, based on a determination that no additional privileges were provided to any users. This was not seen as sufficiently useful for the added complexity.

When creating directories, historical versions of `cp` use the mode of the source directory, plus read, write, and search bits for the owner, as modified by the file mode creation mask. This is done so that `cp` can copy trees where the user has read permission, but the owner does not. A side effect is that if the file creation mask denies the owner permissions, `cp` will fail. Also, once the copy is done, historical versions of `cp` set the permissions on the created directory to be the same as the source directory, unmodified by the file creation mask.

This behavior has been modified so that `cp` will always be able to create the contents of the directory, regardless of the file creation mask. After the copy is done, the permissions are set to be the same as the source directory, as modified by the file creation mask. This latter change from historical behavior is to prevent users from accidentally creating directories with permissions beyond those they would normally set and for consistency with the behavior of `cp` in creating files.

It is not a requirement that `cp` detect attempts to copy a file to itself; however, implementations are strongly encouraged to do so. Historical implementations have detected the attempt in most cases, which is probably all that is needed.

There are two methods of copying subtrees in this standard. The other method is described as part of the `pax` utility (see 4.48). Both methods are historical practice. The `cp` utility provides a simpler, more intuitive interface, while `pax` offers a finer granularity of control. Each provides additional functionality to the other; in particular, `pax` maintains the hard-link structure of the hierarchy, while `cp` does not. It is the intention of the working group that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

The wording allowing `cp` to copy a directory to implementation-defined file types not specified by POSIX.1 {8} is provided so that implementations supporting symbolic links are not required to prohibit copying directories to symbolic links. Other extensions to POSIX.1 {8} file types may need to use this loophole as well.

4.14 `cut` — Cut out selected fields of each line of a file

4.14.1 Synopsis

`cut` `−b` *list* [`−n`] [*file* . . .]

`cut` `−c` *list* [*file* . . .]

`cut` `−f` *list* [`−d` *delim*] [`−s`] [*file* . . .]

4.14.2 Description

The `cut` utility shall 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.

4.14.3 Options

The `cut` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The option-argument *list* (see options `−b`, `−c`, and `−f` below) shall be a comma- 2
separated list or `<blank>`-separated list of positive numbers and ranges. Ranges 2
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 shall be supported by the implementation:

`−b` *list* Cut based on a *list* of bytes. Each selected byte shall be output unless the `−n` option is also specified. It shall not be an error to select bytes not present in the input line.

`−c` *list* Cut based on a *list* of characters. Each selected character shall be output. It shall not be 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 shall be output. Output fields shall be separated by a single occurrence of the field delimiter character. Lines with no field delimiters shall be passed through intact, unless `−s` is specified. It shall not be 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) shall be modified as follows:

> If the byte selected by *low* is not the first byte of a character, *low* shall 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* shall 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 shall be dropped from *list* for that input line without causing an error.

Each element in list of the form *low−* shall be treated as above with *high* set to the the number of bytes in the current line, not including the terminating `<newline>` character. Each element in list of the form *−high* shall be treated as above with *low* set to 1. Each element in list of the form *num* (a single number) shall 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 shall be passed through untouched.

4.14.4 Operands

The following operands shall be supported by the implementation:

file A pathname of an input file. If no *file* operands are specified, or if a *file* operand is −, the standard input shall be used.

4.14.5 External Influences

4.14.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is −. See Input Files.

4.14.5.2 Input Files

The input files shall be text files, except that line lengths shall be unlimited.

4.14.5.3 Environment Variables

The following environment variables shall affect the execution of `cut`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.14.5.4 Asynchronous Events

Default.

4.14.6 External Effects

4.14.6.1 Standard Output

The `cut` utility output shall 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>*

4.14.6.2 Standard Error

Used only for diagnostic messages.

4.14.6.3 Output Files

None.

4.14.7 Extended Description

None.

4.14.8 Exit Status

The `cut` utility shall exit with one of the following values:

- 0 All input files were output successfully.
- >0 An error occurred.

4.14.9 Consequences of Errors

Default.

4.14.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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 through last.

The *low*−*high* forms are not always equivalent when used with `-b` and `-n` and multibyte 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 standard 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
```

History of Decisions Made

Some historical implementations do not count `<backspace>` characters in determining character counts with the `–c` option. This may be useful for using `cut` for processing `nroff` output. It was deliberately decided not to have the `–c` option treat either `<backspace>` or `<tab>` characters in any special fashion. The `fold` utility does treat these characters specially.

Unlike other utilities, some historical implementations of `cut` exit after not finding an input file, rather than continuing to process the remaining *file* operands. This behavior is prohibited by this standard, where only the exit status is affected by this problem.

The behavior of `cut` when provided with either mutually exclusive options or options that do not make sense together has been deliberately left unspecified in favor of global wording in Section 2.

The traditional `cut` utility has worked in an environment where bytes and characters were equivalent (modulo `<backspace>` and `<tab>` processing in some implementations). In the extended world of multibyte 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`.)

4.15 `date` — Write the date and time

4.15.1 Synopsis

`date [−u] [+`*format* `]`

4.15.2 Description

The `date` utility shall write the date and time to standard output. By default, the current date and time shall be written. If an operand beginning with `+` is specified, the output format of `date` shall be controlled by the field descriptors and other text in the operand.

4.15.3 Options

The `date` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−u` Perform operations as if the **TZ** environment variable was set to the string `UTC0`, or its equivalent historical value of `GMT0`. Otherwise, `date` shall use the time zone indicated by the **TZ** environment variable or the system default if that variable is not set. 2

4.15.4 Operands

When the format is specified, each field descriptor shall be replaced in the standard output by its corresponding value. All other characters shall be copied to the output without change. The output shall be always 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>` fill).
`%h`	A synonym for `%b`.

%H	Hour (24-hour clock) as a decimal number (00-23).
%I	Hour (12-hour clock) as a decimal number (01-12).
%j	Day of the year as a decimal number (001-366).
%m	Month as a decimal number (01-12).
%M	Minute as a decimal number (00-59).
%n	A <newline> character.
%p	Locale's equivalent of either AM or PM.
%r	12-Hour clock time (01-12) using the *AM/PM* notation; in the POSIX Locale, this shall be equivalent to "%I:%M:%S %p".
%S	Seconds as a decimal number (00-61).
%t	A <tab> character.
%T	24-Hour clock time (00-23) in the format *HH:MM:SS*.
%U	Week number of the year (Sunday as the first day of the week) as a decimal number (00-53).
%w	Weekday as a decimal number [0 (Sunday)-6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number (00-53).
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year (offset from %C) as a decimal number (00-99).
%Y	Year with century as a decimal number.
%Z	Time-zone name, or no characters if no time zone is determinable.
%%	A <percent-sign> character.

See the LC_TIME description in 2.5.2.5 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 2.5.2.5). If the corresponding keyword (see era, era_year, era_d_fmt, and alt_digits in 2.5.2.5) is not specified or not supported for the current locale, the unmodified field descriptor value shall be used.

%Ec	Locale's alternate appropriate date and time representation.
%EC	The name of the base year (period) in the locale's alternate representation.
%Ex	Locale's alternate date representation.
%Ey	Offset from %EC (year only) in the locale's alternate representation.
%EY	Full alternate year representation.
%Od	Day of month using the locale's alternate numeric symbols.
%Oe	Day of month using the locale's alternate numeric symbols.
%OH	Hour (24-hour clock) using the locale's alternate numeric symbols.

`%OI`	Hour (12-hour clock) using the locale's alternate numeric symbols.
`%Om`	Month using the locale's alternate numeric symbols.
`%OM`	Minutes using the locale's alternate numeric symbols.
`%OS`	Seconds using the locale's alternate numeric symbols.
`%OU`	Week number of the year (Sunday as the first day of the week) using the locale's alternate numeric symbols.
`%Ow`	Weekday as number in the locale's alternate representation (Sunday = 0).
`%OW`	Week number of the year (Monday as the first day of the week) using the locale's alternate numeric symbols.
`%Oy`	Year (offset from `%C`) in alternate representation.

4.15.5 External Influences

4.15.5.1 Standard Input

None.

4.15.5.2 Input Files

None.

4.15.5.3 Environment Variables

The following environment variables shall affect the execution of `date`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

LC_TIME This variable shall determine the format and contents of date and time strings written by `date`.

TZ This variable shall specify the time zone 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 time zone is used.

4.15.5.4 Asynchronous Events

Default.

4.15.6 External Effects

4.15.6.1 Standard Output

When no formatting operand is specified, the output in the POSIX Locale shall be equivalent to specifying

```
date "+%a %b %e %H:%M:%S %Z %Y"
```

4.15.6.2 Standard Error

Used only for diagnostic messages.

4.15.6.3 Output Files

None.

4.15.7 Extended Description

None.

4.15.8 Exit Status

The `date` utility shall exit with one of the following values:

0 The date was written successfully.

>0 An error occurred.

4.15.9 Consequences of Errors

Default.

4.15.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The option for setting the date and time was not included. It is normally a system administration option, which is outside the scope of POSIX.2.

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: 11/21/87
TIME: 13:36:16

$ date "+TIME: %r"
TIME: 01:36:32 PM
```

Field descriptors are of unspecified format when not in the POSIX Locale. Some of them can contain `<newline>`s 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 capitalization 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` (alternate 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 standard was approved in century 19, not the twentieth (let's hope). 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 alternate 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 alternate numerical symbols, such as Kanji or Hindi digits, or ordinal number representation.

Non-European locales, whether they use Latin digits in computational items or 2
not, often have local forms of the digits for use in date formats. This is not totally 2
unknown even in Europe; a variant of dates uses Roman numerals for the 2
months: the third day of September 1991 would be written as 3.IX.1991. In 2
Japan, Kanji digits are regularly used for dates; in Arabic-speaking countries, 2
Hindi digits are used. The `%d`, `%e`, `%H`, `%I`, `%m`, `%S`, `%U`, `%w`, `%W`, and `%y` field descrip- 2
tors always return the date/time field in Latin digits (i.e., 0 through 9). The `%O` 2
modifier was introduced to support the use for display purposes of non-Latin 2
digits. In the LC_TIME category in `localedef`, the optional `alt_digits` key- 2
word is intended for this purpose. As an example, assume the following (partial) 2
`localedef` source: 2

```
alt_digits  "";"I";"II";"III";"IV";"V";"VI";"VII";"VIII" \      2
            "IX";"X";"XI";"XII"                                  2
d_fmt       "%e.%Om.%Y"                                          2
```

With the above date, the command 2

```
date "+x"                                                        2
```

would yield ''3IX.1991.'' With the same `d_fmt`, but without the `alt_digits`, the 2
command would yield ''3.9.1991.'' 2

History of Decisions Made

Some of the new options for formatting are from the C Standard {7}. The `-u` option was introduced to allow portable access to Coordinated Universal Time
(UTC). The string `GMT0` is allowed as an equivalent **TZ** value to be compatible 1
with all of the systems using the BSD implementation, where this option originated.

The `%e` format field descriptor (adopted from System V) was added because the C Standard {7} descriptors did not provide any way to produce the historical default `date` output during the first nine days of any month.

4.16 **dd** — Convert and copy a file

4.16.1 Synopsis

`dd` [*operand* ...]

4.16.2 Description

The `dd` utility shall copy the specified input file to the specified output file with possible conversions using specific input and output block sizes. It shall read the input one block at a time, using the specified input block size; it then shall process the block of data actually returned, which could be smaller than the requested block size. It shall 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` or `noerror` are requested, the data returned from each input block shall 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 shall be the same size as the input block. If the `bs=`*expr* operand is not specified, or a conversion other than `sync` or `noerror` is requested, the input shall be processed and collected into full-sized output blocks until the end of the input is reached.

The processing order shall be 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 shall be appended to the input data up to the specified size. The remaining conversions and output shall 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 shall 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 shall be swapped. If there are an odd number of bytes in the input block, the results are unspecified.

(5) Any remaining conversions (`block`, `unblock`, `lcase`, and `ucase`) shall be performed. These conversions shall 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 shall be aggregated into output blocks of the specified size. After the end of input is reached, any remaining output shall 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.

4.16.3 Options

None.

4.16.4 Operands

All of the operands shall be processed before any input is read. The following operands shall be supported by the implementation:

`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 shall 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 shall be to preserve the blocks in the output file over which `dd` seeks, but no other portion of the output file shall 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 shall 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 2
shall 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 2
value of zero, using `block` or `unblock` produces unspecified 2
results. 2

`skip=`*n* Skip *n* input blocks (using the specified input block size) before starting to copy. On seekable files, the implementation shall read the blocks or seek past them; on nonseekable files, the blocks shall be read and the data shall be discarded.

`seek=`*n* Skip *n* blocks (using the specified output block size) from beginning of output file before copying. On nonseekable files, existing blocks shall 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 shall seek 2
to the specified offset or read the blocks as described for nonseekable files.

`count=`*n* Copy only *n* input blocks.

`conv=`*value* `[,`*value* `...]`
Where *value* s are comma-separated symbols from the following list.

- `block` Treat the input as a sequence of `<newline>`-terminated or end-of-file-terminated variable length records independent of the input block boundaries. Each record shall be converted to a record with a fixed length specified by the conversion block size. Any `<newline>` shall be removed from the input line; `<space>`s shall be appended to lines that are shorter than their conversion block size to fill the block. Lines that are longer than the conversion block size shall be truncated to the largest number of characters that will fit into that size; the number of truncated lines shall be reported (see Standard Error 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, delete all trailing `<space>`s, and append a `<newline>`.

- `lcase` Map uppercase characters specified by the LC_CTYPE keyword `tolower` to the corresponding lowercase character. Characters for which no mapping is specified shall not be modified by this conversion.

  The `lcase` and `ucase` symbols are mutually exclusive.

- `ucase` Map lowercase characters specified by the LC_CTYPE keyword `toupper` to the corresponding uppercase character. Characters for which no mapping is specified shall not be modified by this conversion.

- `swab` Swap every pair of input bytes.

- `noerror` Do not stop processing on an input error. When an input error occurs, a diagnostic message shall be written on standard error, followed by the current input and output block counts in the same format as used at completion (see Standard Error). If the `sync` conversion is specified, the missing input shall be replaced with null bytes and processed normally; otherwise, the input block shall 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 `ibs=` buffer, appending null bytes. 2

The behavior is unspecified if operands other than `conv=` are specified more than once.

For the `bs=`, `cbs=`, `ibs=`, and `obs=` operands, the application shall 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; or

(4) two or more positive decimal numbers (with or without `k` or `b`) separated by `x`, specifying the product of the indicated values.

4.16.5 External Influences

4.16.5.1 Standard Input

If no `if=` operand is specified, the standard input shall be used. See Input Files.

4.16.5.2 Input Files

The input file can be any file type.

4.16.5.3 Environment Variables

The following environment variables shall affect the execution of `dd`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), the classification of characters as upper- or lowercase, and the mapping of characters from one case to the other.

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

4.16.5.4 Asynchronous Events

For SIGINT, the `dd` utility shall write status information to standard error before exiting. It shall take the standard action for all other signals; see 2.11.5.4.

4.16.6 External Effects

4.16.6.1 Standard Output

If no `of=` operand is specified, the standard output shall be used. The nature of the output depends on the operands selected.

4.16.6.2 Standard Error

On completion, `dd` shall write the number of input and output blocks to standard error. In the POSIX Locale the following formats shall 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 shall be written to standard error. In the POSIX Locale, the format shall be:

`"%u truncated %s\n"`, *<number of truncated blocks>*, `"block"` [if *<number of truncated blocks>* is one] `"blocks"` [otherwise]

Diagnostic messages may also be written to standard error.

4.16.6.3 Output Files

If the `of=` operand is used, the output shall be the same as described in Standard Output.

4.16.7 Extended Description

None.

4.16.8 Exit Status

The dd utility shall exit with one of the following values:

0 The input file was copied successfully.

>0 An error occurred.

4.16.9 Consequences of Errors

If an input error is detected and the noerror conversion has not been specified, any partial output block shall be written to the output file, a diagnostic message shall be written, and the copy operation shall be discontinued. If some other error is detected, a diagnostic message shall be written and the copy operation shall be discontinued.

4.16.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The input and output block size can be specified to take advantage of raw physical I/O.

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.

A suggested implementation technique for conv=noerror,sync is to zero the input buffer before each read and to write the contents of the input buffer to the output even after an error. In this manner, any data transferred to the input buffer before the error was detected will be preserved. Another point is that a failed read on a regular file or a disk will generally not increment the file offset, and dd must then seek past the block on which the error occurred; otherwise, the input error will occur repetitively. When the input is a magnetic tape, however, the tape will normally have passed the block containing the error when the error is reported, and thus no seek is necessary.

History of Decisions Made

Table 4-4 – ASCII to EBCDIC Conversion

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**
0000	0000	0001	0002	0003	0067	0055	0056	0057
0010	0026	0005	0045	0013	0014	0015	0016	0017
0020	0020	0021	0022	0023	0074	0075	0062	0046
0030	0030	0031	0077	0047	0034	0035	0036	0037
0040	0100	0132	0177	0173	0133	0154	0120	0175
0050	0115	0135	0134	0116	0153	0140	0113	0141
0060	0360	0361	0362	0363	0364	0365	0366	0367
0070	0370	0371	0172	0136	0114	0176	0156	0157
0100	0174	0301	0302	0303	0304	0305	0306	0307
0110	0310	0311	0321	0322	0323	0324	0325	0326
0120	0327	0330	0331	0342	0343	0344	0345	0346
0130	0347	0350	0351	0255	0340	0275	0232	0155
0140	0171	0201	0202	0203	0204	0205	0206	0207
0150	0210	0211	0221	0222	0223	0224	0225	0226
0160	0227	0230	0231	0242	0243	0244	0245	0246
0170	0247	0250	0251	0300	0117	0320	0137	0007
0200	0040	0041	0042	0043	0044	0025	0006	0027
0210	0050	0051	0052	0053	0054	0011	0012	0033
0220	0060	0061	0032	0063	0064	0065	0066	0010
0230	0070	0071	0072	0073	0004	0024	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

The Options subclause is listed as ‘‘None’’because there are no options recognized by historical dd utilities. Certainly, many of the operands could have been designed to use the Utility Syntax Guidelines, which would have resulted in the classic hyphenated option letters. In this version of this standard, dd retains its curious JCL-like syntax due to the large number of applications that depend on the historical implementation. ‘‘Fixing’’ the interface would cause an excessive compatibility problem. However, due to interest in the international community, the developers of the standard have agreed to provide an alternative syntax for

Table 4-5 – ASCII to IBM EBCDIC Conversion

	0	1	2	3	4	5	6	7
0000	0000	0001	0002	0003	0067	0055	0056	0057
0010	0026	0005	0045	0013	0014	0015	0016	0017
0020	0020	0021	0022	0023	0074	0075	0062	0046
0030	0030	0031	0077	0047	0034	0035	0036	0037
0040	0100	0132	0177	0173	0133	0154	0120	0175
0050	0115	0135	0134	0116	0153	0140	0113	0141
0060	0360	0361	0362	0363	0364	0365	0366	0367
0070	0370	0371	0172	0136	0114	0176	0156	0157
0100	0174	0301	0302	0303	0304	0305	0306	0307
0110	0310	0311	0321	0322	0323	0324	0325	0326
0120	0327	0330	0331	0342	0343	0344	0345	0346
0130	0347	0350	0351	0255	0340	0275	0137	0155
0140	0171	0201	0202	0203	0204	0205	0206	0207
0150	0210	0211	0221	0222	0223	0224	0225	0226
0160	0227	0230	0231	0242	0243	0244	0245	0246
0170	0247	0250	0251	0300	0117	0320	0241	0007
0200	0040	0041	0042	0043	0044	0025	0006	0027
0210	0050	0051	0052	0053	0054	0011	0012	0033
0220	0060	0061	0032	0063	0064	0065	0066	0010
0230	0070	0071	0072	0073	0004	0024	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

the next version of this standard that conforms to the spirit of the Utility Syntax Guidelines. This new syntax will be accompanied by the existing syntax, marked as obsolescent. System implementors are encouraged to develop and promulgate a new syntax for `dd`, perhaps using a different utility name, that can be adopted for the next version of this standard.

The default `ibs=` and `obs=` sizes are specified as 512 bytes because there are existing (largely portable) scripts that assume these values. If they were left unspecified, very strange results could occur if an implementation chose an odd block size.

Historical implementations of `dd` used *creat*() when processing `of=file`. This makes the `seek=` operand unusable except on special files. More recent BSD-based implementations use *open*() (without O_TRUNC) instead of *creat*(), but fail to delete output file contents after the data copied. Since balloting showed a desire to make this behavior available, the `conv=notrunc` feature was added.

The `w` multiplier, (historically meaning *word*), is used in System V to mean 2 and in 4.2BSD to mean 4. Since *word* is inherently nonportable, its use is not supported by POSIX.2.

All references to US ASCII and to conversions to/from IBM and EBCDIC were removed in preparation for this document's acceptance by the international community. Implementations are free to have such conversions as extensions, using the `ascii`, `ibm`, and `ebcdic` keywords. However, in the interest of promoting consistency of implementation, the original material from an early draft has been restored to the rationale as an example:

> In the two tables, the conversions from ASCII to either standard EBCDIC (Table 4-4) or the IBM version of EBCDIC (Table 4-5) are shown. The differences between the two tables are underlined. 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 tables are understood to match recent System V conversion algorithms and there have been reports that earlier System V versions and the BSD version do not always conform to these; however, representatives of the BSD development group have agreed that a future version of their system will use these tables for consistency with System V.
>
> The `cbs` operand is required if any of the `ascii`, `ebcdic`, or `ibm` operands are 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>`s 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>`s are added.

The `block` and `unblock` keywords are from historical BSD practice.

Early drafts only allowed two numbers separated by `x` to be used in a product when specifying `bs=`, `cbs=`, `ibs=`, and `obs=` sizes. This was changed to reflect the historical practice of allowing multiple numbers in the product as provided by Version 7 and all releases of System V and BSD.

4.17 diff — Compare two files

4.17.1 Synopsis

```
diff [ −c | −e | −C n ] [−br] file1 file2
```

4.17.2 Description

The diff utility shall 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 shall be produced if the files are identical.

4.17.3 Options

The diff utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−b Cause trailing <blank>s to be ignored and other strings of <blank>s 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* shall be interpreted as a positive decimal integer).

−e Produce output in a form suitable as input for the ed utility (see 4.20), which can then be used to convert *file1* into *file2*.

−r Apply diff recursively to files and directories of the same name when *file1* and *file2* are both directories.

4.17.4 Operands

The following operands shall be supported by the implementation:

file1
file2 A pathname of a file be compared. If either the *file1* or *file2* operand is −, the standard input shall be used in its place.

If both *file1* and *file2* are directories, diff shall not compare block special files, character special files, or FIFO special files to any files and shall not compare regular files to directories. The system documentation shall specify the behavior of diff on implementation-specific file types not specified by POSIX.1 {8} when found in directories. Further details are as specified in 4.17.6.1.1.

If only one of *file1* and *file2* is a directory, diff shall be applied to the nondirectory file and the file contained in the directory file with a filename that is the same as the last component of the nondirectory file.

4.17.5 External Influences

4.17.5.1 Standard Input

The standard input shall be used only if one of the *file1* or *file2* operands references standard input. See Input Files.

4.17.5.2 Input Files

The input files shall be text files.

4.17.5.3 Environment Variables

The following environment variables shall affect the execution of `diff`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

LC_TIME This variable shall determine the locale for affecting the format of file time stamps written with the `−C` and `−c` options.

TZ This variable shall determine the locale for affecting the time zone used for calculating file time stamps written with the `−C` and `−c` options.

4.17.5.4 Asynchronous Events

Default.

4.17.6 External Effects

4.17.6.1 Standard Output

4.17.6.1.1 `diff` Directory Comparison Format

If both *file1* and *file2* are directories, the following output formats shall be used.

In the POSIX Locale, each file that is present in only one directory shall 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 shall be used to write the differences.

All directory pathnames listed in this subclause shall be relative to the original command line arguments. All other names of files listed in this subclause shall be filenames (pathname components).

4.17.6.1.2 `diff` Default Output Format

The default (without `−e`, `−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 shall pertain to *file1* ; those after shall 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 shall 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"

4.17.6.1.3 **diff −e Output Format**

With the −e option, a script shall be produced that shall, when provided as input to ed (see 4.20), 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 shall be used in this script. Text line(s), except those consisting of the single character period (.), shall be output as they appear in the file.

4.17.6.1.4 **diff −c or −C Output Format**

With the −c or −C option, the output format shall consist of affected lines along with surrounding lines of context. The affected lines shall 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, shall be written before and after the affected lines. With the −C option, the user can specify how many lines of context shall be written. The exact format follows.

The name and last modification time of each file shall 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 shall be the pathname of the corresponding file being compared. The pathname written for standard input is unspecified.

In the POSIX Locale, each *<time stamp>* field shall be equivalent to the output from the following command:

`date "+%a %b %e %T %Y"`

without the trailing <newline>, 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 shall be applied for every set of changes.

First, the range of lines in *file1* shall 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) shall be written. Unaffected lines shall be written in the following format:

`"ΔΔ%s"`, *<unaffected_line>*

Deleted lines shall be written as:

`"-Δ%s"`, *<deleted_line>*

Changed lines shall be written as:

`"!Δ%s"`, *<changed_line>*

Next, the range of lines in *file2* shall be written in the following format:

`"--- %d,%d ----\n"`, *<beginning line number>* , *<ending line number>*

Then, lines of context and changed lines shall be written as described in the previous formats. Lines added from *file2* shall be written in the following format:

`"+Δ%s"`, *<added_line>*

4.17.6.2 Standard Error

Used only for diagnostic messages.

4.17.6.3 Output Files

None.

4.17.7 Extended Description

None.

4.17.8 Exit Status

The `diff` utility shall exit with one of the following values:

0	No differences were found.
1	Differences were found.
>1	An error occurred.

4.17.9 Consequences of Errors

Default.

4.17.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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.

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
```

History of Decisions Made

The −h option was removed because it was insufficiently specified and it does not add to application portability.

Current implementations employ algorithms that do not always produce a minimum list of differences; the current language about making every effort is the best the standard can do, as there is no metric that could be employed to judge the quality of implementations against any and all file contents. The statement ‘‘This list should be minimal’’ clearly implies that implementations are not expected to provide the following output when comparing two 100-line files that differ in only one character on a single line:

```
1,100c1,100
all 100 lines from file1 preceded with "< "
---
all 100 lines from file2 preceded with "> "
```

The ‘‘Only in’’ messages required by this standard when the −r option is specified, is not used by most historical implementations if the −e option is also specified. It is required here because it provides useful information that must be provided to update a target directory hierarchy to match a source hierarchy. The ‘‘Common subdirectories’’ messages are written by System V and 4.3BSD when the −r option is specified. They are allowed here, but are not required because they are reporting on something that is the same, not reporting a difference, and are not needed to update a target hierarchy.

The `-c` option, which writes output in a format using lines of context, has been included. The format is useful for a variety of reasons, among them being much improved readability, and the ability to understand difference changes when the target file has line numbers that differ from another similar, but slightly different, copy. An important utility, `patch`, which has proved itself indispensable to the USENET community, often only works with difference listings using the context format. The BSD version of `-c` takes an optional argument specifying the amount of context. Rather than overloading `-c` and breaking the Utility Syntax Guidelines for `diff`, the working group decided to add a separate option for specifying a context diff with a specified amount of context (`-C`). Also, the format for context diffs was extended slightly in 4.3BSD to allow multiple changes that are within context lines from each other to be merged together. The output format contains an additional four asterisks after the range of affected lines in the first filename. This was to provide a flag for old programs (like old versions of `patch`) that only understand the old context format. The version of context described here does not require that multiple changes within context lines be merged, but does not prohibit it either. The extension is upward compatible, so any vendors that wish to retain the old version of `diff` can do so by just adding the extra four asterisks (that is, utilities that currently use `diff` and understand the new merged format will also understand the old unmerged format, but not vice-versa).

The substitute command was added as an additional format for the `-e` option. This was added to provide implementations a way to fix the classic ‘‘dot alone on a line’’ bug present in many versions of `diff`. Since many implementations have fixed this bug the working group decided not to standardize broken behavior, but rather, provide the necessary tool for fixing the bug. One way to fix this bug is to output two periods whenever a lone period is needed, then terminate the append command with a period, and then use the substitute command to convert the two periods into one period.

The `-f` flag was not included as it provides no additional functionality over the `-e` option.

The BSD-derived `-r` option was added to provide a mechanism for using `diff` to compare two file system trees. This behavior is useful, is standard practice on all BSD-derived systems, and is not easily reproducible with the `find` utility.

The requirement that `diff` not compare files in some circumstances, even though they have the same name, was added in response to ballot objections and digging further into the actual output of historical implementations. The message specified here is already in use when a directory is being compared to a nondirectory. It is extended here to preclude the problems arising from running into FIFOs and other files that would cause `diff` to hang waiting for input with no indication to the user that `diff` was hung. In most common usage, `diff -r` should indicate differences in the file hierarchies, not the difference of contents of devices pointed to by the hierarchies.

Many early implementations of `diff` require seekable files. Since POSIX.1 {8} supports named pipes, the working group decided that such a restriction was unreasonable. Note also that the allowed file name − almost always refers to a pipe.

No directory search order is being specified in 4.17.6.1.1. The historical ordering is, in fact, not optimal, in that it prints out all of the differences at the current level, including the statements about all common subdirectories before recursing into those subdirectories.

The message 2

`"diff %s %s %s\n"`, <*diff_options*> , <*filename1*> , <*filename2*> 2

does not vary by locale because it is the representation of a command, not an 2
English sentence. 2

4.18 `dirname` — Return directory portion of pathname

4.18.1 Synopsis

`dirname` *string*

4.18.2 Description

The *string* operand shall be treated as a pathname, as defined in 2.2.2.102. The string *string* shall 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) through (5).

(2) If *string* consists entirely of slash characters, *string* shall be set to a single slash character. In this case, skip steps (3) through (8).

(3) If there are any trailing slash characters in *string*, they shall be removed.

(4) If there are no slash characters remaining in *string*, *string* shall be set to a single period character. In this case, skip steps (5) through (8).

(5) If there are any trailing nonslash characters in *string*, they shall be removed.

(6) If the remaining *string* is `//`, it is implementation defined whether steps (7) and (8) are skipped or processed.

(7) If there are any trailing slash characters in *string*, they shall be removed.

(8) If the remaining *string* is empty, *string* shall be set to a single slash character.

The resulting string shall be written to standard output.

4.18.3 Options

None.

4.18.4 Operands

The following operand shall be supported by the implementation:

string A string.

4.18.5 External Influences

4.18.5.1 Standard Input

None.

4.18.5.2 Input Files

None.

4.18.5.3 Environment Variables

The following environment variables shall affect the execution of `dirname`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.18.5.4 Asynchronous Events

Default.

4.18.6 External Effects

4.18.6.1 Standard Output

The `dirname` utility shall write a line to the standard output in the following format:

`"%s\n"`, *<resulting string>*

4.18.6.2 Standard Error

Used only for diagnostic messages.

4.18.6.3 Output Files

None.

4.18.7 Extended Description

None.

4.18.8 Exit Status

The `dirname` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.18.9 Consequences of Errors

Default.

4.18.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `dirname` utility originated in System III. It has evolved through the System V releases to a version that matches the requirements specified in this description in System V Release 3.

4.3BSD and earlier versions did not include `dirname`.

Table 4-6 indicates the results required for some invocations of `dirname`.

Table 4-6 – `dirname` 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`

History of Decisions Made

The behaviors of `basename` and `dirname` in this standard have been coordinated so that when *string* is a valid pathname

```
$(basename "string")
```

would be a valid filename for the file in the directory

```
$(dirname "string")
```

This would not work for the versions of these utilities in earlier drafts due to the way processing of trailing slashes was specified. Consideration was given to leaving processing unspecified if there were trailing slashes, but this cannot be done; the POSIX.1 {8} definition of pathname allows trailing slashes. The `basename` and `dirname` utilities have to specify consistent handling for all valid pathnames.

Since the definition of *pathname* in 2.2.2.102 specifies implementation-defined behavior for pathnames starting with two slash characters, Draft 11 has been changed to specify similar implementation-defined behavior for the `basename` and `dirname` utilities. On implementations where the pathname `//` is always treated the same as the pathname `/`, the functionality required by Draft 10 meets all of the Draft 11 requirements.

4.19 `echo` — Write arguments to standard output

4.19.1 Synopsis

`echo` [*string* ...]

4.19.2 Description

The `echo` utility shall write its arguments to standard output, followed by a `<newline>` character. If there are no arguments, only the `<newline>` character shall be written.

4.19.3 Options

The `echo` utility shall not recognize the `--` argument in the manner specified by utility syntax guideline 10 in 2.10.2; `--` shall be recognized as a string operand.

Implementations need not support any options.

4.19.4 Operands

The following operands shall be supported by the implementation:

string A string to be written to standard output. If the first operand is `"-n"` or if any of the operands contain a backslash (\) character, the results are implementation defined.

4.19.5 External Influences

4.19.5.1 Standard Input

None.

4.19.5.2 Input Files

None.

4.19.5.3 Environment Variables

The following environment variables shall affect the execution of `echo`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_MESSAGES This variable shall determine the language in which diagnostic messages should be written.

4.19.5.4 Asynchronous Events

Default.

4.19.6 External Effects

4.19.6.1 Standard Output

The `echo` utility arguments shall be separated by single `<space>`s and a `<newline>` character shall follow the last argument.

4.19.6.2 Standard Error

Used only for diagnostic messages.

4.19.6.3 Output Files

None.

4.19.7 Extended Description

None.

4.19.8 Exit Status

The `echo` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

4.19.9 Consequences of Errors

Default.

4.19.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

As specified by this standard, echo writes its arguments in the simplest of ways. The two different historical versions of echo vary in fatal incompatible ways.

The BSD echo checks the first argument for the string "−n", which causes it to suppress the <newline> character that would otherwise follow the final argument in the output.

The System V echo does not support any options, but allows escape sequences within its operands:

\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 are 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 1-, 2-, or 3-digit octal number *num*.

It is not possible to use echo portably across these two implementations unless both −n (as the first argument) and escape sequences are omitted.

The printf utility (see 4.50) can be used to portably emulate any of the traditional behaviors of the echo utility as follows:

— The System V 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
```

The echo utility does not support utility syntax guideline 10 because existing applications depend on echo to echo *all* of its arguments, except for the −n option in the BSD version.

New applications are encouraged to use `printf` instead of `echo`. The `echo` utility has not been made obsolescent because of its extremely widespread use in existing applications.

History of Decisions Made

In Draft 8, an attempt was made to merge the extensions of BSD and System V, supporting both `−n` and escape sequences. During initial ballot resolution, a `−e` option was proposed to enable the escape conventions. Both attempts failed, as there are historical scripts that would be broken by any attempt at reconciliation. Therefore, in Draft 9 only the simplest version of `echo` is presented. Implementation-defined extensions on BSD and System V will keep historical applications content. Portable applications that wish to do prompting without `<newline>`s or that could possibly be expecting to echo a `"−n"`, should use the new `printf` utility (see 4.50), derived from the Ninth Edition.

The **LC_CTYPE** variable is not cited because `echo`, as specified here, does not need to understand the characters in its arguments. The System V and BSD implementations might need to be sensitive to it because of their extensions.

4.20 ed — Edit text

4.20.1 Synopsis

`ed` [`−p` *string*] [`−s`] [*file*]

Obsolescent Version :

`ed` [`−p` *string*] [`−`] [*file*]

4.20.2 Description

The `ed` utility is a line-oriented text editor that shall use two modes: *command mode* and *input mode* . In command mode the input characters shall be interpreted as commands, and in input mode they shall be interpreted as text. See 4.20.7.

4.20.3 Options

The `ed` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except for its nonstandard usage of – in the obsolescent version.

The following options shall be supported by the implementation:

`-p` *string* Use *string* as the prompt string when in command mode. By default, there shall be no prompt string.

`-s` Suppress the writing of byte counts by `e`, `E`, `r`, and `w` commands and of the `!` prompt after a `!`*command* .

`-` (Obsolescent.) Equivalent to the `-s` option.

4.20.4 Operands

The following operand shall be supported by the implementation:

file If the *file* argument is given, `ed` shall simulate an `e` command on the file named by the pathname, *file*, before accepting commands from the standard input.

4.20.5 External Influences

4.20.5.1 Standard Input

The standard input shall be a text file consisting of commands, as described in 4.20.7.

4.20.5.2 Input Files

The input files shall be text files.

4.20.5.3 Environment Variables

The following environment variables shall affect the execution of `ed`:

HOME This variable shall determine the pathname of the user's home directory.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), the behavior of character classes within regular expressions.

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

4.20.5.4 Asynchronous Events

The `ed` utility shall take the standard action for all signals (see 2.11.5.4), with the following exceptions:

SIGINT The `ed` utility shall interrupt its current activity, write the string

```
"?\n"
```

to standard output, and return to command mode (see 4.20.7).

SIGHUP If the buffer is not empty and has changed since the last write, the `ed` utility shall attempt to write a copy of the buffer in a file. First, the file named `ed.hup` in the current directory shall be used; if that fails, the file named `ed.hup` in the directory named by the **HOME** environment variable shall be used. In any case, the `ed` utility shall exit without returning to command mode.

4.20.6 External Effects

4.20.6.1 Standard Output

Various editing commands and the prompting feature (see `−p`) write to standard output, as described in 4.20.7.

4.20.6.2 Standard Error

Used only for diagnostic messages.

4.20.6.3 Output Files

The output files shall be text files whose formats are dependent on the editing commands given.

4.20.7 Extended Description

The `ed` utility shall operate on a copy of the file it is editing; changes made to the copy shall have no effect on the file until a `w` (write) command is given. The copy of the text is called the *buffer* in this clause, although no attempt is made to imply a specific implementation.

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 shall 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 shall be recognized; all input is merely collected. Input mode is terminated by entering a line consisting of two characters: a period (`.`) followed by a <newline>. This line is not considered part of the input text.

4.20.7.1 ed Regular Expressions

The `ed` utility shall support basic regular expressions, as described in 2.8.3. 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>. A null RE shall be 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.

4.20.7.2 ed Addresses

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 shall not modify the current line number.

Addresses are constructed as follows:

(1) The character `.` (period) shall address the current line.

(2) The character `$` shall address the last line of the buffer.

(3) A positive decimal number *n* shall address the *n*-th line of the buffer. The first line in the buffer is line number 1.

(4) `'`*x* shall address the line marked with the mark name character *x*, which shall be a lowercase letter from the portable character set. Lines can be marked with the `k` command described in 4.20.7.3.13.

(5) An RE enclosed by slashes (`/`) shall address 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. [As stated in 4.20.7.1, an address consisting of a null RE delimited by slashes (`//`) shall address the next line containing the last RE encountered.] If necessary, the search shall 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 `\/` shall represent a literal slash instead of the RE delimiter.

(6) An RE enclosed in question-marks (`?`) shall address 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 `\?` shall represent 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 shall be added to or subtracted from the address, respectively. As a consequence of this rule and of rule (8) immediately above, the address `−` shall refer to the line preceding the current line. Moreover, trailing `+` and `−` characters shall have a cumulative effect, so `−−` shall refer to the current line number less 2.

(10) A comma (`,`) shall stand for the address pair `1,$`, while a semicolon (`;`) shall stand for the pair `.,$`.

Commands require zero, one, or two addresses. Commands that require no addresses shall regard the presence of an address as an error. Commands that accept one or two addresses assume default addresses when no addresses are given, as described in 4.20.7.3. If one address is given to a command that allows two addresses, the command shall 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 (`.`) shall be set to the first address, and only then shall 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 shall correspond to a line that does not precede, in the buffer, the line corresponding to the first address.

4.20.7.3 `ed` Commands

In the following list of `ed` commands, the default addresses are shown in parentheses. The number of addresses shown in the default shall be 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 shall be exe- 1
cuted and then the new current line shall be written as described below under the 1
`l`, `n`, and `p` commands. When an `l`, `n`, or `p` suffix is used with an `l`, `n`, or `p` command, the command shall 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) shall either write just the current line or shall write it twice—once as specified for `p` and once as specified for `l`. Also, the `g`, `G`, `v`, and `V` commands shall take a command as a parameter.

Each address component can be preceded by zero or more `<blank>`s. The command letter can be preceded by zero or more `<blank>`s. If a suffix letter (`l`, `n`, or `p`) is given, it shall immediately follow the command.

The `e`, `E`, `f`, `r`, and `w` commands shall take an optional *file* parameter, separated from the command letter by one or more `<blank>`s.

If changes have been made in the buffer since the last `w` command that wrote the entire buffer, `ed` shall warn the user if an attempt is made to destroy the editor buffer via the `e` or `q` commands. The `ed` utility shall write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the `H` command) to standard output and shall continue in command mode with the current line number unchanged. If the `e` or `q` command is repeated with no intervening command, it shall take effect.

If an end-of-file is detected on standard input when a command is expected, the `ed` utility shall act as if a `q` command had been entered.

If the closing delimiter of an RE or of a replacement string (e.g., `/`) in a `g`, `G`, `s`, `v`, or `V` command would be the last character before a `<newline>`, that delimiter can be omitted, in which case the addressed line shall be 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` shall write the string:

```
"?\n"
```

(followed by an explanatory message if *help mode* has been enabled via the `H` command) to standard output and shall continue in command mode with the current line number unchanged.

4.20.7.3.1 Append Command

Synopsis : `(.)a`
<text>
`.`

The *append* command shall read the given text and append it after the addressed line; the current line number shall become the address of the last inserted line, or, if there were none, the addressed line. Address 0 shall be valid for this command: it shall cause the ‘‘appended’’ text to be placed at the beginning of the buffer.

4.20.7.3.2 Change Command

Synopsis : `(.,.)c` 1
<text>
`.`

The *change* command shall delete the addressed lines, then accept input text that replaces these lines; the current line shall 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 shall be set to the address of the new last line; if no lines remain in the buffer, the current line number shall be set to zero.

4.20.7.3.3 Delete Command

Synopsis : `(.,.)d`

The *delete* command shall delete the addressed lines from the buffer. The address of the line after the last line deleted shall become the current line number; if the lines deleted were originally at the end of the buffer, the current line number shall be set to the address of the new last line; if no lines remain in the buffer, the current line number shall be set to zero.

4.20.7.3.4 Edit Command

Synopsis : `e` [*file*]

The *edit* command shall delete the entire contents of the buffer and then read in the file named by the pathname *file*. The current line number shall be set to the address of the last line of the buffer. If no pathname is given, the currently remembered pathname, if any, shall be used (see the `f` command). The number of bytes read shall be written to standard output, unless the `−s` option was specified, in the following format:

`"%d\n"`, *<number of bytes read>*

The name *file* shall 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 shall be taken to be a shell command line whose output is to be read. Such a shell command line shall not be remembered as the current *file*. All marks shall 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 shall be warned, as described previously.

4.20.7.3.5 Edit Without Checking Command

Synopsis : `E` [*file*]

The *Edit* command shall possess all properties and restrictions of the `e` command except that the editor shall not check to see if any changes have been made to the buffer since the last `w` command.

4.20.7.3.6 File-Name Command

Synopsis : `f` [*file*]

If *file* is given, the file-name command shall change the currently remembered pathname to *file*; whether the name is changed or not, it then shall write the (possibly new) currently remembered pathname to the standard output in the following format:

`"%s\n"`, *<pathname>*

The current line number shall be unchanged.

4.20.7.3.7 Global Command

Synopsis : `(1,$)g/`*RE/command list*

In the *global* command, the first step shall be to mark every line that matches the given *RE*. Then, for every such line, the given *command list* shall be executed with the current line number set to the address of that line. When the `g` command completes, the current line number shall have the value assigned by the last command in the command list. If there were no matching lines, the current line number shall not be changed. A single command or the first of a list of commands shall appear on the same line as the global command. All lines of a multiline list except the last line shall 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* shall be 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.

4.20.7.3.8 Interactive Global Command

Synopsis : `(1,$)G/`*RE/*

In the *interactive global* command, the first step shall be to mark every line that matches the given *RE*. Then, for every such line, that line shall be written, the current line number shall 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 shall be executed. A `<newline>` shall act as a null command (causing no action to be taken on the current line); an `&` shall cause the reexecution of the most recent nonnull 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 shall be the value set by the last command successfully executed. (Note that the last command successfully executed shall 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 shall 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.

4.20.7.3.9 Help Command

Synopsis : `h`

The *help* command shall write a short message to standard output that explains the reason for the most recent `?` notification. The current line number shall be unchanged.

4.20.7.3.10 Help-Mode Command

Synopsis : `H`

The *Help* command shall cause `ed` to enter a mode in which help messages (see the `h` command) shall be written to standard output for all subsequent `?` notifications. The `H` command alternately shall turn this mode on and off; it shall be initially off. If the help-mode is being turned on, the `H` command also shall explain the previous `?` notification, if there was one. The current line number shall be unchanged.

4.20.7.3.11 Insert Command

Synopsis : `(.)i`
<text>
`.`

The *insert* command shall insert the given text before the addressed line; `.` shall 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 shall be invalid for this command.

4.20.7.3.12 Join Command

Synopsis : `(.,.+1)j`

The *join* command shall join contiguous lines by removing the appropriate `<newline>` characters. If exactly one address is given, this command shall do nothing. If lines are joined, the current line number shall be set to the address of the joined line; otherwise, the current line number shall be unchanged.

4.20.7.3.13 Mark Command

Synopsis : `(.)k`*x*

The *mark* command shall mark the addressed line with name *x*, which shall be a lowercase letter from the portable character set. The address `'`*x* then shall refer to this line; the current line number shall be unchanged.

4.20.7.3.14 List Command

Synopsis : `(.,.)l`

The *list* command shall write to standard output the addressed lines in a visually unambiguous form. The characters listed in Table 2-15 (see 2.12) shall be written as the corresponding escape sequence. Nonprintable characters not in Table 2-15 shall be written as one three-digit octal number (with a preceding `<backslash>`) 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 nonprintable characters is implementation defined.

Long lines shall be folded, with the point of folding indicated by writing `<backslash><newline>`; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line shall 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 shall be set to the address of the last line written.

4.20.7.3.15 Move Command

Synopsis : `(.,.)m`*address*

The *move* command shall reposition the addressed line(s) after the line addressed by *address* . Address 0 shall be valid for *address* and cause the addressed line(s) to be moved to the beginning of the buffer. It shall be an error if address *address* falls within the range of moved lines. The current line number shall be set to the address of the last line moved.

4.20.7.3.16 Number Command

Synopsis : `(.,.)n`

The *number* command shall write to standard output the addressed lines, preceding each line by its line number and a `<tab>` character; the current line number shall be set to the address of the last line written. The `n` command can be appended to any other command other than `e`, `E`, `f`, `q`, `Q`, `r`, `w`, or `!`.

4.20.7.3.17 Print Command

Synopsis : `(.,.)p`

The *print* command shall write to standard output the addressed lines; the current line number shall be set to the address of the last line written. The `p` command can be appended to any other command other than `e`, `E`, `f`, `q`, `Q`, `r`, `w`, or `!`.

4.20.7.3.18 Prompt Command

Synopsis : `P`

The *Prompt* command shall cause `ed` to prompt with an asterisk (*) (or *string* , if `−p` is specified) for all subsequent commands. The `P` command alternately shall turn this mode on and off; it shall be initially on if the `−p` option is specified, otherwise off. The current line number shall be unchanged.

4.20.7.3.19 Quit Command

Synopsis : `q`

The *quit* command shall cause `ed` to exit. If the buffer has changed since the last time the entire buffer was written, the user shall be warned, as described previously.

4.20.7.3.20 Quit Without Checking Command

Synopsis : `Q`

The *Quit* command shall cause `ed` to exit without checking if changes have been made in the buffer since the last `w` command.

4.20.7.3.21 Read Command

Synopsis : `($)r` [*file*]

The *read* command shall read in the file named by the pathname *file* and append it after the addressed line. If no *file* argument is given, the currently remembered pathname, if any, shall be used (see `e` and `f` commands). The currently remembered pathname shall not be changed unless there is no remembered pathname. Address 0 shall be valid for `r` and shall cause 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 shall be written to standard output in the following format:

`"%d\n"`, <*number of bytes read*>

The current line number shall be set to the address of the last line read in. If *file* is replaced by `!`, the rest of the line shall be taken to be a shell command line whose output is to be read. Such a shell command line shall not be remembered as the current pathname.

4.20.7.3.22 Substitute Command

Synopsis : `(.,.)s/` *RE/replacement/flags*

The *substitute* command shall search each addressed line for an occurrence of the specified RE and replace either the first or all (nonoverlapped) 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 shall be set to the address of the last line on which a substitution occurred.

An ampersand (&) appearing in the *replacement* shall 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, shall be replaced by the text matched by the corresponding backreference expression (see 2.8.3.3). When the character `%` is the only character in the *replacement* , the *replacement* used in the most recent substitute command shall be used as the *replacement* in the current substitute command; if there was no previous substitute command, the use of `%` in this manner shall be an error. The `%` shall lose its special meaning when it is in a replacement string of more than one character or is preceded by a backslash.

A line can be split by substituting a `<newline>` character into it. The application 1
shall escape the `<newline>` in the *replacement* by preceding it by backslash. 1
Such substitution cannot be done as part of a `g` or `v` command list. The current line number shall be set to the address of the last line on which a substitution is performed. If no substitution is performed, the current line number shall be unchanged. If a line is split, a substitution shall be considered to have been performed on each of the new lines for the purpose of determining the new current line number. A substitution shall be considered to have been performed even if the replacement string is identical to the string that it replaces.

The value of *flags* shall 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 nonoverlapping 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 shall 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 shall 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 shall be written in the format specified for the `p` command.

4.20.7.3.23 Copy Command

Synopsis : `(.,.)t`*address*

The `t` command shall be equivalent to the `m` command, except that a copy of the addressed lines shall be placed after address *address* (which can be 0); the current line number shall be set to the address of the last line added.

4.20.7.3.24 Undo Command

Synopsis : `u`

The *undo* command shall nullify 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 1
command shall be ''undone'' as a single change; if no changes were made by the 1
global command (such as with `g/`*RE*`/p`), the `u` command shall have no effect. The 1
current line number shall be set to the value it had immediately before the command being undone started.

4.20.7.3.25 Global Non-Matched Command

Synopsis : `(1,$)v/`*RE/command list*

This command shall be equivalent to the global command `g` except that the lines that are marked during the first step shall be those that do not match the RE.

4.20.7.3.26 Interactive Global Not-Matched Command

Synopsis : `(1,$)V/`*RE/*

This command shall be equivalent to the interactive global command `G` except that the lines that are marked during the first step shall be those that do not match the RE.

4.20.7.3.27 Write Command

Synopsis : `(1,$)w [`*file*`]`

The *write* command shall write the addressed lines into the file named by the pathname *file*. The command shall create the file, if it does not exist, or shall replace the contents of the existing file. The currently remembered pathname shall not be changed unless there is no remembered pathname. If no pathname is given, the currently remembered pathname, if any, shall be used (see `e` and `f` commands); the current line number shall be unchanged. If the command is successful, the number of bytes written shall 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 shall be taken to be a shell command line whose standard input shall be the addressed lines. Such a shell command line
shall not be remembered as the current pathname. This usage of the write com- 1
mand with `!` shall not be considered as a ''last `w` command that wrote the entire 1
buffer,'' as described previously; thus, this alone shall not prevent the warning to 1
the user if an attempt is made to destroy the editor buffer via the `e` or `q` com- 1
mands. 1

4.20.7.3.28 Line Number Command

Synopsis : `($)=`

The line number of the addressed line shall be written to standard output in the following format:

`"%d\n"`, <*line number*>

The current line number shall be unchanged by this command.

4.20.7.3.29 Shell Escape Command

Synopsis : `!`*command*

The remainder of the line after the `!` shall 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 `%` shall be replaced with the remembered pathname; if a `!` appears as the first character of the command, it shall be replaced with the text of the previous shell command executed via `!`. Thus, `!!` shall repeat the pre-
vious `!`*command* . If any replacements of `%` and/or `!` are performed, the modified 2
line shall be written to the standard output before *command* is executed. The `!` 2
command shall write 2

`"!\n"`

to standard output upon completion, unless the `-s` option is specified. The current line number shall be unchanged.

4.20.7.3.30 Null Command

Synopsis : `(.+1)`

An address alone on a line shall cause the addressed line to be written. A `<newline>` alone shall be equivalent to `.+1p`. The current line number shall be set to the address of the written line.

4.20.8 Exit Status

The `ed` utility shall exit with one of the following values:

0 Successful completion without any file or command errors.

>0 An error occurred.

4.20.9 Consequences of Errors

When an error in the input script is encountered, or when an error is detected 1
that is a consequence of the data (not) present in the file or due to an external 1
condition such as a read or write error: 1

— If the standard input is a terminal device file, all input shall be flushed, 2
and a new command read. 2

— If the standard input is a regular file, `ed` shall terminate with a nonzero 2
exit status. 2

4.20.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Some historical implementations contained a bug that allowed a single period to be entered in input mode as `<backslash>` `<period>` `<newline>`. This is not allowed by the POSIX.2 `ed` because there is no description of escaping any of the characters in input mode; backslashes are entered into the buffer exactly as typed. The typical method of entering a single period has been to precede it with another character and then use the substitute command to delete that character.

Because of the extremely terse nature of the default error messages, the prudent 1
script writer will begin the `ed` input commands with an `H` command, so that if any 1
errors do occur at least some clue as to the cause will be made available. 1

History of Decisions Made

The initial description of this utility was adapted from the *SVID*. It contains some features not found in Version 7 or BSD-derived systems. Some of the differences between the POSIX.2 and BSD `ed` utilities include, but need not be limited to:

— The BSD `-` option does not suppress the `!` prompt after a `!` command.

— BSD does not support the special meanings of the `%` and `!` characters within a `!` command.

— BSD does not support the *addresses* `;` and `,`.

— BSD allows the command/suffix pairs `pp`, `ll`, etc., which are unspecified in POSIX.2.

— BSD does not support the `!` character part of the `e`, `r`, or `w` commands.

- A failed `g` command in BSD sets the line number to the last line searched if there are no matches.
- BSD does not default the command list to the `p` command.
- BSD does not support the `G`, `h`, `H`, `n`, or `V` commands.
- On BSD, if there is no inserted text, the insert command changes the current line to the referenced line −1; i.e., the line before the specified line.
- On BSD, the join command with only a single address changes the current line to that address.
- BSD does not support the `P` command; moreover, in BSD it is synonymous with the `p` command.
- BSD does not support the *undo* of the commands `j`, `m`, `r`, `s`, or `t`.
- The BSD `ed` commands `W`, `wq`, and `z` are not present in POSIX.2.

The `-s` option was added to allow the functionality of the `-` option in a manner compatible with the Utility Syntax Guidelines. It is the intent of the working group that portable applications use the `-s` option, and that in the future the `-` option be removed from the standard.

Prior to Draft 8 there was a limit, {ED_FILE_MAX}, which described the historical limitations of some `ed` utilities in their handling of large files; some of these have had problems with files in the >100 KB range. It was this limitation that prompted much of the desire to include a `split` command in the standard. Since this limit was removed, the standard requires that implementations document the file size limits imposed by `ed` in the conformance document. The limit {ED_LINE_MAX} was also removed; therefore, the global limit {LINE_MAX} is used for input and output lines.

The `\{`*m*`,`*n*`\}` notation was removed from the description of regular expressions because this functionality is now described in 2.8.3.

The manner in which the `l` command writes nonprintable characters was changed to avoid the historical backspace-overstrike method. On video display terminals, the overstrike is ambiguous because most terminals simply replace overstruck characters, making the `l` format not useful for its intended purpose of unambiguously understanding the content of the line. The historical backslash escapes were also ambiguous. (The string `"a\0011"` could represent a line containing those six characters or a line containing the three characters `'a'`, a byte with a binary value of 1, and a `'1'`.) In the format required here, a backslash appearing in the line will be written as `"\\"` so that the output is truly unambiguous. The method of marking the ends of lines was adopted from the `ex` editor (see the User Portability Extension) and is required for any line ending in `<space>`s; the `$` is placed on all lines so that a real `$` at the end of a line cannot be misinterpreted.

Systems with bytes too large to fit into three octal digits must devise other means of displaying nonprintable characters. Consideration was given to requiring that the number of octal digits be large enough to hold a byte, but this seemed to be too confusing for applications on the vast majority of systems where three digits are adequate. It would be theoretically possible for the application to use the `getconf` utility to find out the {CHAR_BIT} value and deal with such an algorithm; however, there is really no portable way that an application can use the octal values of the bytes across various coded character sets anyway, so the additional specification did not seem worth the effort.

The description of how a NUL is written was removed. The NUL character cannot be in text files, and the standard should not dictate behavior in the case of undefined, erroneous input.

The text requiring filenames accepted by the `E`, `e`, `R`, and `r` commands to be patterns was removed due to balloting objections that this was undesirable and not existing practice.

The `−p` option in Drafts 8 and 9 said that it only worked when standard input was associated with a terminal device. This has been changed to conform to existing implementations, thereby allowing applications to interpose themselves between a user and the `ed` utility.

The form of the substitute command that uses the *n* suffix was limited to the first 512 matches in a previous draft (where this was described incorrectly as ‘‘backreferencing’’). This limit has been removed because there is no reason an editor processing lines of {LINE_MAX} length should have this restriction. The command `s/x/X/2047` should be able to substitute the 2047th occurrence of `x` on a line.

The use of printing commands with printing suffixes (such as `pn`, `lp`, etc.) was made unspecified because BSD-based systems allow this, whereas System V does not.

Some BSD-based systems exit immediately upon receipt of end-of-file if all of the lines in the file had been deleted. Since POSIX.2 refers to the `q` command in this instance, such behavior is not allowed.

Some historical implementations returned exit status zero even if command errors had occurred; this is not allowed by POSIX.2.

4.21 env — Set environment for command invocation

4.21.1 Synopsis

`env [−i] [`*name*=*value* `] . . . [`*utility* `[`*argument* `. . . ]]`

Obsolescent Version :

`env [−] [`*name*=*value* `] . . . [`*utility* `[`*argument* `. . . ]]`

4.21.2 Description

The `env` utility shall 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 shall be passed to *utility* .

If no *utility* operand is specified, the resulting environment shall be written to the standard output, with one *name* =*value* pair per line.

4.21.3 Options

The `env` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except for its nonstandard usage of −, which is obsolescent.

The following options shall be supported by the implementation:

`−i` Invoke *utility* with exactly the environment specified by the arguments; the inherited environment shall be ignored completely.

`−` (Obsolescent.) Equivalent to the `−i` option.

4.21.4 Operands

The following operands shall be supported by the implementation:

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 3.14, the results are undefined.

argument A string to pass as an argument for the invoked utility.

4.21.5 External Influences

4.21.5.1 Standard Input

None.

4.21.5.2 Input Files

None.

4.21.5.3 Environment Variables

The following environment variables shall affect the execution of `env`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

PATH This variable shall determine the location of the *utility*, as described in 2.6. If **PATH** is specified as a *name*=*value* operand to `env`, the *value* given shall be used in the search for *utility*.

4.21.5.4 Asynchronous Events

Default.

4.21.6 External Effects

4.21.6.1 Standard Output

If no *utility* operand is specified, each *name*=*value* pair in the resulting environment shall be written in the form:

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

If the *utility* operand is specified, the `env` utility shall not write to standard output.

4.21.6.2 Standard Error

Used only for diagnostic messages.

4.21.6.3 Output Files

None.

4.21.7 Extended Description

None.

4.21.8 Exit Status

If the *utility* utility is invoked, the exit status of `env` shall be the exit status of *utility;* otherwise, the `env` utility shall exit with one of the following values:

0	The `env` utility completed successfully.
1−125	An error occurred in the `env` utility. 1
126	The utility specified by *utility* was found but could not be invoked. 1
127	The utility specified by *utility* could not be found. 1

4.21.9 Consequences of Errors

Default.

4.21.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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`.

As with all other utilities that invoke other utilities, the standard only specifies what `env` does with standard input, standard output, standard error, input files, and output files. If a utility is executed, it is not constrained by `env`'s specification of input and output.

The `command`, `env`, `nohup`, and `xargs` utilities have been specified to use exit
code 127 if an error occurs so that applications can distinguish ''failure to find a 1
utility'' from ''invoked utility exited with an error indication.'' The value 127 was 1
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 con-
fused with termination due to receipt of a signal. The value 126 was chosen in a 1
similar manner to indicate that the utility could be found, but not invoked. Some 1
scripts produce meaningful error messages differentiating the 126 and 127 cases. 1
The distinction between exit codes 126 and 127 is based on KornShell practice 2
that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 2
126 when any attempt to *exec* the utility fails for any other reason. 2

History of Decisions Made

The `−i` option was added to allow the functionality of the − option in a manner compatible with the Utility Syntax Guidelines. It is the intent of the working group that portable applications use the `−i` option, and that in the future the − option be removed from the standard.

Historical implementations of the `env` utility use *execvp* () or *execlp* () (see POSIX.1 {8} 3.1.2) 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. Implementations are free to invoke a shell instead of using one of the *exec* family of routines, but if they do, they must be sure to escape any characters with special meaning to the shell so that the user does not have to be aware of the difference.

Some have suggested that `env` is redundant since the same effect is achieved by:

```
name=value ... utility [argument ...]
```

The example is equivalent to `env` when an environment variable is being added to the environment of the command, but not when the environment is being set to the given value. The `env` utility also writes out the current environment if invoked without arguments. There is sufficient functionality beyond what the example provides to justify inclusion of `env`.

4.22 `expr` — Evaluate arguments as an expression

4.22.1 Synopsis

`expr` *operand* . . .

4.22.2 Description

The `expr` utility shall evaluate an expression and write the result to standard output.

4.22.3 Options

None.

4.22.4 Operands

The single expression evaluated by `expr` shall be formed from the operands, as described in 4.22.7. Each of the expression operator symbols:

`(   )   |   &   =   >   >=   <   <=   !=   +   −   *   /   %   :`

and the symbols *integer* and *string* in the table shall be provided by the application as separate arguments to `expr`.

4.22.5 External Influences

4.22.5.1 Standard Input

None.

4.22.5.2 Input Files

None.

4.22.5.3 Environment Variables

The following environment variables shall affect the execution of `expr`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions and by the string comparison operators.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes within regular expressions.

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

4.22.5.4 Asynchronous Events

Default.

4.22.6 External Effects

4.22.6.1 Standard Output

The `expr` utility shall evaluate the expression and write the result to standard output. The character `'0'` shall be written to indicate a zero value and nothing shall be written to indicate a null string.

4.22.6.2 Standard Error

Used only for diagnostic messages.

4.22.6.3 Output Files

None.

4.22.7 Extended Description

The formation of the expression to be evaluated is shown in Table 4-7. 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 in Table 4-7 are listed in order of increasing precedence, with equal-precedence operators grouped between horizontal lines. All of the operators shall be left-associative.

Table 4-7 – `expr` Expressions

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 shall be 1 if the specified relation is true, or 0 if the relation 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 4.22.7.1.
(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 4.22.7.2.

4.22.7.1 Matching Expression

The `':'` matching operator shall compare the string resulting from the evaluation of *expr1* with the regular expression pattern resulting from the evaluation of *expr2* . Regular expression syntax shall be that defined in 2.8.3 (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 shall be matched by the regular expression) and, therefore, it is unspecified whether ˆ is a special character in that context. Usually, the matching operator shall 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` shall be returned (see 2.8.3.3).

4.22.7.2 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 4.22.4.

The use of string arguments `length`, `substr`, `index`, or `match` produces unspecified results.

4.22.8 Exit Status

The `expr` utility shall exit with one of the following values:

- 0 If the *expression* evaluates to neither null nor zero.
- 1 If the *expression* evaluates to null or zero.
- 2 For invalid *expression* s.
- >2 An error occurred.

4.22.9 Consequences of Errors

Default.

4.22.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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>`s 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`; the utility was retained by POSIX.2 as acknowledgment of the many historical shell scripts that use it. Newly written scripts should avoid `expr` in favor of the new features within the shell.

The following command

```
a=$(expr $a + 1)
```

adds 1 to the variable **a**. A new application should use 1

```
a=$(($a+1))
``` 
1

The following command, for $a equal to either /usr/abc/file or just file:

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

returns the last segment of a pathname (i.e., 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**.

Usage Warning : 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 standard 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 (see 2.10.2) to protect its operands if there is any chance the first operand might be a negative integer (or any string with a leading minus).

History of Decisions Made

In an earlier draft, Extended Regular Expressions were used in the matching expression syntax. This was changed to the Basic variety to avoid breaking historical applications.

The use of a leading circumflex in the regular expression is unspecified because many historical implementations have treated it as special, despite their system documentation. For example,

```
expr foo : ^foo                expr ^foo : ^foo
```

return 3 and 0, respectively, on those systems; their documentation would imply the reverse. Thus, the anchoring condition is left unspecified to avoid breaking historical scripts relying on this undocumented feature.

4.23 `false` — Return false value

4.23.1 Synopsis

```
false
```

4.23.2 Description

The `false` utility shall return with a nonzero exit code.

4.23.3 Options

None.

4.23.4 Operands

None.

4.23.5 External Influences

4.23.5.1 Standard Input

None.

4.23.5.2 Input Files

None.

4.23.5.3 Environment Variables

None.

4.23.5.4 Asynchronous Events

Default.

4.23.6 External Effects

4.23.6.1 Standard Output

None.

4.23.6.2 Standard Error

None.

4.23.6.3 Output Files

None.

4.23.7 Extended Description

None.

4.23.8 Exit Status

The `false` utility always shall exit with a value other than zero.

4.23.9 Consequences of Errors

Default.

4.23.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `false` utility is typically used in shell control structures like `while`.

4.24 `find` — Find files

4.24.1 Synopsis

`find` *path* ... [*operand_expression* ...]

4.24.2 Description

The `find` utility shall recursively descend the directory hierarchy from each file specified by *path*, evaluating a Boolean expression composed of the primaries described in 4.24.4 for each file encountered.

The `find` utility shall be able to descend to arbitrary depths in a file hierarchy and shall not fail due to path length limitations (unless a path operand specified by the application exceeds {PATH_MAX} requirements).

The `find` utility requires that the underlying system provides information equivalent to the *st_dev*, *st_mode*, *st_nlink*, *st_uid*, *st_gid*, *st_size*, *st_atime*, *st_mtime*, and *st_ctime* members of *struct stat* described by POSIX.1 {8} 5.6 and conforming to the *file times update* definition in 2.2.2.69.

4.24.3 Options

None.

4.24.4 Operands

The following operands shall be supported by the implementation:

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 shall 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 shall 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*

Implementations shall recognize the following primaries: *Editor's Note: These primaries have been sorted alphabetically, without diff marks.*

`−atime` *n* The primary shall evaluate as true if the file access time subtracted from the initialization time is $n-1$ to n multiples of 24 hours. The initialization time shall 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 shall evaluate as true if the time of last change of file status information subtracted from the initialization time is $n-1$ to *n* multiples of 24 hours. The initialization time shall 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.

`−depth` The primary always shall evaluate as true; it shall 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 shall be acted on after the directory itself. If any `−depth` primary is specified, it shall apply to the entire expression even if the `−depth` primary would not normally be evaluated.

`-exec` *utility_name* [*argument* ...] `;`
The primary shall evaluate as true if the invoked utility *utility_name* returns a zero value as exit status. The end of the primary expression shall be punctuated by a semicolon. A *utility_name* or *argument* containing only the two characters `{}` shall 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 defined 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* shall 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 3.14, the results are undefined.

`−group` *gname* The primary shall 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* shall be interpreted as a group ID.

`−links` *n* The primary shall evaluate as true if the file has *n* links.

`−mtime` *n* The primary shall evaluate as true if the file modification time subtracted from the initialization time is $n-1$ to *n* multiples of 24 hours. The initialization time shall 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.

`–name` *pattern* The primary shall evaluate as true if the basename of the filename being examined matches *pattern* using the pattern matching notation described in 3.13.

`–newer` *file* The primary shall 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*.

`–nogroup` The primary shall evaluate as true if the file belongs to a group ID for which the POSIX.1 {8} *getgrgid*() (or equivalent) function returns **NULL**.

`–nouser` The primary shall evaluate as true if the file belongs to a user ID for which the POSIX.1 {8} *getpwuid*() (or equivalent) function returns **NULL**.

`-ok` *utility_name* [*argument* ...] `;`
The `–ok` primary shall be equivalent to `–exec`, except that `find` shall request affirmation of the invocation of *utility_name* using the current file as an argument by writing to standard error as, described in 4.24.6.2. If the response on standard input is affirmative, the utility shall be invoked. Otherwise, the command shall not be invoked and the value of the `–ok` operand shall be false.

`-perm [-]`*mode* The *mode* argument is used to represent file mode bits. It shall be identical in format to the *symbolic_mode* operand described in 4.7, and shall be interpreted as follows. To start, a template shall be assumed with all file mode bits cleared. An *op* symbol of `+` shall set the appropriate mode bits in the template; `-` shall clear the appropriate bits; `=` shall set the appropriate mode bits, without regard to the contents of process's file mode creation mask. The *op* symbol of `-` cannot be the first character of *mode*.

If the hyphen is omitted, the primary shall 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 shall evaluate as true if at least all the bits in the resulting template are set in the file permission bits.

`-perm [-]`*onum* (Obsolescent.) If the hyphen is omitted, the primary shall 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 shall be compared. (See the description of the octal *mode* in 4.7.) Otherwise, if *onum* is prefixed by a hyphen, the primary shall evaluate as true if at least all of the bits specified in *onum* that are also set it the octal mask 07777 are set.

`−print` The primary always shall evaluate as true; it shall cause the current pathname to be written to standard output.

`−prune` The primary always shall evaluate as true; it shall cause `find` not to descend the current pathname if it is a directory. If the `−depth` primary is specified, the `−prune` primary shall have no effect.

`-size` *n*`[c]` The primary shall 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 shall be in bytes.

`−type` *c* The primary shall 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.

`−user` *uname* The primary shall 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* shall be interpreted as a user ID.

`−xdev` The primary always shall evaluate as true; it shall cause `find` not to continue descending past directories that have a different device ID (*st_dev* , see POSIX .1 {8} 5.6.2). If any `−xdev` primary is specified, it shall apply to the entire expression even if the `−xdev` 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 shall be implied by the juxtaposition of two primaries or made explicit by the optional `−a` operator. The second expression shall not be evaluated if the first expression is false.

expression `−o` *expression*
Alternation of primaries; the OR operator. The second expression shall not be evaluated if the first expression is true.

If no *expression* is present, `−print` shall be used as the expression. Otherwise, if the given expression does not contain any of the primaries `−exec`, `−ok`, or `−print`, the given expression shall be effectively replaced by:

`(` *given_expression* `) -print`

The `−user`, `−group`, and `−newer` primaries each shall evaluate their respective arguments only once.

4.24.5 External Influences

4.24.5.1 Standard Input

If the `–ok` primary is used, the response shall be read from the standard input. An entire line shall be read as the response. Otherwise, the standard input shall not be used.

4.24.5.2 Input Files

None.

4.24.5.3 Environment Variables

The following environment variables shall affect the execution of `find`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements used in the pattern matching notation for the `–name` option and in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments), the behavior of character classes within the pattern matching notation used for the `–name` option, and the behavior 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 This variable shall determine the processing of affirmative responses and the language in which messages should be written.

PATH This variable shall determine the location of the *utility_name* for the `–exec` and `–ok` primaries, as described in 2.6.

4.24.5.4 Asynchronous Events

Default.

4.24.6 External Effects

4.24.6.1 Standard Output

The `-print` primary shall cause the current pathnames to be written to standard output. The format shall be:

`"%s\n"`, <*path*>

4.24.6.2 Standard Error

The `-ok` primary shall 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 shall be `?`. The exact format used is unspecified.

Otherwise, the standard error shall be used only for diagnostic messages.

4.24.6.3 Output Files

None.

4.24.7 Extended Description

None.

4.24.8 Exit Status

The `find` utility shall exit with one of the following values:

- 0 All *path* operands were traversed successfully.
- >0 An error occurred.

4.24.9 Consequences of Errors

Default.

4.24.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

When used in operands, pattern matching notation, semicolons, opening parentheses, and closing parentheses are special to the shell and must be quoted (see 3.2).

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.

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.

The `−prune` primary was adopted from later releases of 4.3 BSD and the third edi- 1
tion of the *SVID* . The following command recursively prints pathnames of all files 1
in the current directory and below, but skips directories named SCCS and files in them.

```
find . -name SCCS -prune -o -print
```

The following command behaves as in the previous example, but prints the names of the SCCS directories.

```
find . -print -name SCCS -prune
```

The following command is roughly equivalent to the `−nt` extension to `test`: 1

```
if [ -n "$(find file1 -prune -newer file2)" ]; then          2
          printf %s\\n "file1 is newer than file2"           2
fi                                                           1
```

History of Decisions Made

The historical `−a` operator is kept as an optional operator for compatibility with existing shell scripts even though it is redundant with expression concatenation.

The symbolic means of specifying file permission bits, based on `chmod`, was added in response to numerous balloting objections that `find` was the only remaining utility to not support this method. The warning about a leading *Op* of `-` is to avoid 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. 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 POSIX .1 {8} or POSIX. 2, applications must not assume that it actually refers to the traditional sticky bit.

The descriptions of how the - modifier on the *mode* and *onum* arguments to the `−perm` primary affects processing has been documented here to match the way it behaves in practice on historical BSD and System V implementations. System V and BSD documentation both describe it in terms of checking additional bits; in fact, it uses the same bits, but checks for having at least all of the matching bits set instead of having exactly the matching bits set.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified, because:

(1) Implementations may desire more descriptive prompts than those used on historical implementations.

(2) Since the traditional prompt strings do not terminate with `<newline>`s, there is no portable way for another program to interact with the prompts of this utility via pipes.

Therefore, an application using this prompting option relies on the system to provide the most suitable dialogue directly with the user, based on the general guidelines specified.

The `−name` *file* operand was changed to use the shell pattern matching notation so that `find` is consistent with other utilities using pattern matching.

For the `−type` *c* operand, implementors of symbolic links should consider `l` (the letter ell) for symbolic links. Implementations that support sockets also use `−type s` for sockets. Implementations planning to add options to allow find to follow symbolic links or treat them as special files, should consider using `−follow` as used in BSD and System V Release 4 as a guide.

The `−size` operand refers to the size of a file, rather than the number of blocks it 2
may occupy in the file system. The intent is that the POSIX.1 {8} *st_size* field 2
should be used, not the *st_blocks* found in historical implementations. There are 2
at least two reasons for this: 2

— In both System V and BSD, `find` only uses *st_size* in size calculations for 2
the operands specified by POSIX.2. (BSD uses *st_blocks* only when process- 2
ing the `−ls` primary.) 2

— Users will usually be thinking of size in terms of the size of the file in bytes, 2
which is also used by the `ls` utility for the output from the `−l` option. (In 2
both System V in BSD, `ls` uses *st_size* for the `−l` option size field and uses 2
st_blocks for the `ls −s` calculations. POSIX.2 does not specify `ls −s`.) 2

The descriptions of `−atime`, `−ctime`, and `−mtime` were changed from the *SVID*'s description of *n* ‘‘days’’ to ‘‘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. The 1
description is also different in terms of the exact timeframe for the *n* case (versus 1
the +*n* or −*n*), but it matches all known historical implementations. It refers to 1

one 24-hour period in the past, not any time from the beginning of that period to the current time. For example, `−atime 3` is true if the file was accessed any time in the period from 72 to 48 hours ago.

Historical implementations do not modify `{}` when it appears as a substring of an `−exec` or `−ok` *utility_name* or argument string. There have been numerous user requests for this extension, so this standard allows the desired behavior. At least one recent implementation does support this feature, but ran into several problems in managing memory allocation and dealing with multiple occurrences of `{}` in a string while it was being developed, so it is not yet required behavior.

Assuming the presence of `−print` was added at the request of several working group members to correct a historical pitfall that plagues novice users. It is entirely upward compatible from the historical System V `find` utility and should be easy to implement. In its simplest form (`find` *directory*), it could be confused with the historical BSD fast `find`. The BSD developers agree that adding `−print` as a default expression is the right thing to do and believe that the fast `find` functionality should have been/should be provided by a separate utility. They suggest that the new utility be called `locate`.

4.25 fold — Fold lines

4.25.1 Synopsis

`fold [−bs][−w` *width* `][`*file* ... `]`

4.25.2 Description

The `fold` utility is a filter that shall 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 shall be broken by the insertion of a `<newline>` character such that each output line (referred to later in this clause as a segment) is the maximum width possible that does not exceed the specified number of column positions (or bytes). A line shall not be broken in the middle of a character. The behavior 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 shall be treated specially:

`<carriage-return>`
: The current count of line width shall be set to zero. The `fold` utility shall not insert a `<newline>` immediately before or after any `<carriage-return>`.

`<backspace>`
: The current count of line width shall be decremented by one, although the count never shall become negative. The `fold` utility shall not

insert a `<newline>` immediately before or after any `<backspace>`.

`<tab>` Each `<tab>` character encountered shall advance the column position pointer to the next tab stop. Tab stops shall be at each column position *n* such that *n* modulo 8 equals 1.

4.25.3 Options

The `fold` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−b` Count *width* in bytes rather than column positions.

`−s` If a segment of a line contains a `<blank>` within the first *width* column positions (or bytes), break the line after the last such `<blank>` meeting the width constraints. If there is no `<blank>` meeting the requirements, the `−s` option shall 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 shall be 80.

4.25.4 Operands

The following operand shall be supported by the implementation:

file A pathname of a text file to be folded. If no *file* operands are specified, the standard input shall be used.

4.25.5 External Influences

4.25.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.25.5.2 Input Files

If the `−b` option is specified, the input files shall 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 shall be text files.

4.25.5.3 Environment Variables

The following environment variables shall affect the execution of `fold`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte 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 This variable shall determine the language in which messages should be written.

4.25.5.4 Asynchronous Events

Default.

4.25.6 External Effects

4.25.6.1 Standard Output

The standard output shall be a file containing a sequence of characters whose order shall be preserved from the input file(s), possibly with inserted `<newline>` characters.

4.25.6.2 Standard Error

Used only for diagnostic messages.

4.25.6.3 Output Files

None.

4.25.7 Extended Description

None.

4.25.8 Exit Status

The `fold` utility shall exit with one of the following values:

- 0 All input files were processed successfully.
- >0 An error occurred.

4.25.9 Consequences of Errors

Default.

4.25.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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 needs 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.)

Although terminal input in canonical processing mode requires the erase character (frequently set to `<backspace>`) to erase the previous character (not byte or column position), terminal output is not buffered and is extremely difficult, if not impossible, to parse correctly; the interpretation depends entirely on the physical device that will actually display/print/store the output. In all known internationalized implementations, the utilities producing output for mixed column width output assume that a `<backspace>` backs up one column position and outputs enough `<backspace>`s to get back to the start of the character when `<backspace>` is used to provide local line motions to support underlining and emboldening operations. Since `fold` without the `−b` option is dealing with these same constraints, `<backspace>` is always treated as backing up one column position rather than backing up one character.

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
```

History of Decisions Made

Historical versions of the `fold` utility assumed one byte was one character and occupied one column position when written out. This is no longer always true. Since the most common usage of `fold` is believed to be folding long lines for output to limited-length output devices, this capability was preserved as the default case. The `-b` option was added so that applications could `fold` files with arbitrary length lines into text files that could then be processed by the utilities in this standard. Note that although the width for the `-b` option is in bytes, a line will never be split in the middle of a character. (It is unspecified what happens if a width is specified that is too small to hold a single character found in the input followed by a `<newline>`.)

The use of a hyphen as an option to specify standard input was removed from an earlier draft because it adds no functionality and is not historical practice.

The tab stops are hardcoded to be every eighth column to meet historical practice. No new method of specifying other tab stops was invented.

4.26 `getconf` — Get configuration values

4.26.1 Synopsis

`getconf` *system_var*

`getconf` *path_var pathname*

4.26.2 Description

In the first synopsis form, the `getconf` utility shall write to the standard output the value of the variable specified by the *system_var* operand.

In the second synopsis form, the `getconf` utility shall 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 shall be determined as if it were obtained by calling the function from which it is defined to be available by this standard or by POSIX.1 {8} (see Operands). The value shall reflect conditions in the current operating environment.

4.26.3 Options

None.

4.26.4 Operands

The following operands shall be supported by the implementation:

system_var A name of a configuration variable whose value is available from the function defined in 7.8.1 [such as *confstr* () in the C binding], from the POSIX.1 {8} *sysconf* () function, one of the additional POSIX.2 variables described in 7.8.2, to be available from the *sysconf* () function, or a minimum value specified by POSIX.1 {8} or POSIX.2 for one of these variables.

The configuration variables and minimum values listed in the:

— Name column of Table 2-16 (Utility Limit Minimum Values)

— Name column of Table 2-17 (Symbolic Utility Limits)

— Name column of Table 2-18 (Optional Facility Configuration Values)

— Name column of POSIX.1 {8} Table 2-3 (Minimum Values)

— Name column of POSIX.1 {8} Table 2-4 (Run-Time Increasable Values)

— Variable column of POSIX.1 {8} Table 4-2 (Configurable System Variables; except CLK_TCK need not be supported), without the enclosing braces and PATH [corresponding to the *confstr* () name value _CS_PATH] shall be recognized as valid *system_var* operands. The implementation may support additional *system_var* operand values.

path_var A name of a configuration variable whose value is available from the POSIX.1 {8} *pathconf* () function.

The configuration variables listed in the Variable column of the POSIX.1 {8} Table 5-2 (Configurable Pathname Variables), without the enclosing braces, shall be recognized as valid *path_var* operands. The implementation may support additional *path_var* operand values.

pathname A pathname for which the variable specified by *path_var* is to be determined.

4.26.5 External Influences

4.26.5.1 Standard Input

None.

4.26.5.2 Input Files

None.

4.26.5.3 Environment Variables

The following environment variables shall affect the execution of `getconf`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.26.5.4 Asynchronous Events

Default.

4.26.6 External Effects

4.26.6.1 Standard Output

If the specified variable is defined on the system and its value is described to be available from the function in 7.8.1, its value shall be written in the following format:

`"%s\n"`, *<value>*

Otherwise, if the specified variable is defined on the system, its value shall be written in the following format:

`"%d\n"`, *<value>*

If the specified variable is valid, but is undefined on the system, `getconf` shall write using the following format:

```
"undefined\n"
```

If the variable name is invalid or an error occurs, nothing shall be written to standard output.

4.26.6.2 Standard Error

Used only for diagnostic messages.

4.26.6.3 Output Files

None.

4.26.7 Extended Description

None.

4.26.8 Exit Status

The `getconf` utility shall exit with one of the following values:

- 0 The specified variable is valid and information about its current state was written successfully.
- >0 An error occurred.

4.26.9 Consequences of Errors

Default.

4.26.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The original need for this utility, and for the *confstr* () function, was to provide a way of finding the configuration-defined default value for the **PATH** environment variable. Since **PATH** can be modified by the user to include directories that could contain utilities replacing the POSIX.2 standard utilities, shell scripts need a way to determine the system supplied **PATH** environment variable value that contains the correct search path for the standard utilities.

It was later suggested that access to the other variables described here could also be useful to applications.

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=$(getconf PATH_MAX /usr); then                              1
        if [ "$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.

History of Decisions Made

This utility was renamed from `posixconf` during balloting because the new name expresses its purpose more specifically, and does not unduly restrict the scope of application of the utility.

This functionality of this utility would not be adequately subsumed by another command such as

```
grep var /etc/conf
```

because such a strategy would provide correct values for neither those variables that can vary at run-time, nor those that can vary depending on the path.

Previous versions of this utility specified exit status 1 when the specified variable was valid, but not defined on the system. The output string `"undefined"` is now used to specify this case with exit code 0 because so many things depend on an exit code of zero when an invoked utility is successful.

4.27 `getopts` — Parse utility options

4.27.1 Synopsis

`getopts` *optstring name* [*arg* . . .]

4.27.2 Description

The `getopts` utility can be used to retrieve options and option-arguments from a list of parameters. It shall support the utility argument syntax guidelines 3 through 10, inclusive, described in 2.10.2.

Each time it is invoked, the `getopts` utility shall place 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** shall be initialized to 1.

When the option requires an option-argument, the `getopts` utility shall 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** shall be unset. 1

If an option character not contained in the *optstring* operand is found where an option character is expected, the shell variable specified by *name* shall be set to the question-mark (?) character. In this case, if the first character in *optstring* is a colon (`:`), the shell variable **OPTARG** shall be set to the option character found, but no output shall be written to standard error; otherwise, the shell variable **OPTARG** shall be unset and a diagnostic message shall be written to standard error. This condition shall be considered to be an error detected in the way arguments were presented to the invoking application, but shall not be 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* shall be set to the colon character and the shell variable **OPTARG** shall be set to the option character found.

— Otherwise, the shell variable specified by *name* shall be set to the question-mark character, the shell variable **OPTARG** shall be unset, and a diagnostic message shall be written to standard error. This condition shall be considered to be an error detected in the way arguments were presented to the invoking application, but shall not be an error in `getopts` processing; a diagnostic message shall be written as stated, but the exit status shall be zero.

When the end of options is encountered, the `getopts` utility shall exit with a return value greater than zero; the shell variable **OPTIND** shall be set to the index of the first nonoption-argument, where the first `--` argument is considered to be an option-argument if there are no other nonoption-arguments appearing before it, or the value `$#` + 1 if there are no nonoption-arguments; the *name* variable shall be set to the question-mark character. Any of the following shall identify 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** shall be local to the caller of `getopts` and shall not be exported by default.

The shell variable specified by the *name* operand, **OPTIND** , and **OPTARG** shall affect the current shell execution environment; see 3.12.

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.

4.27.3 Options

None.

4.27.4 Operands

The following operands shall be supported by the implementation:

optstring A string containing the option characters recognized by the utility invoking `getopts`. If a character is followed by a colon, the option shall 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` shall 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 recognized if it is not supplied as a separate argument when `getopts` is invoked. [See also the *getopt* () Description in B.7]. The characters question-mark and colon shall 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** shall be stripped of the option character and the `'-'`. The first character in *optstring* shall determine how `getopts` shall behave if an option character is not known or an option-argument is missing. See 4.27.2.

name The name of a shell variable that shall be set by the `getopts` utility to the option character that was found. See 4.27.2.

The `getopts` utility by default shall parse positional parameters passed to the invoking shell procedure. If *arg* s are given, they shall be parsed instead of the positional parameters.

4.27.5 External Influences

4.27.5.1 Standard Input

None.

4.27.5.2 Input Files

None.

4.27.5.3 Environment Variables

The following environment variables shall affect the execution of `getopts`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

OPTIND This variable shall be used by the `getopts` utility as the index of the next argument to be processed.

4.27.5.4 Asynchronous Events

Default.

4.27.6 External Effects

4.27.6.1 Standard Output

None.

4.27.6.2 Standard Error

Whenever an error is detected and the first character in the *optstring* operand is not a colon (`:`), a diagnostic message shall be written to standard error with the 1
following information in an unspecified format: 1

— The invoking program name shall be identified in the message. The invoking program name shall be the value of the shell special parameter `0` (see 3.5.2) 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 shall be identified and the invalid option character shall be identified in the message.

— If an option requiring an option-argument is found, but an option-argument is not found, this error shall be identified and the invalid option character shall be identified in the message.

4.27.6.3 Output Files

None.

4.27.7 Extended Description

None.

4.27.8 Exit Status

The `getopts` utility shall exit with one of the following values:

0 An option, specified or unspecified by *optstring* , was found.

>0 The end of options was encountered or an error occurred.

4.27.9 Consequences of Errors

Default.

4.27.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `getopts` utility was chosen in preference to the `getopt` utility specified in System V because `getopts` handles option-arguments containing `<blank>` characters.

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.

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" "$*"
```

History of Decisions Made

The **OPTARG** variable is not mentioned in the Environment Variables subclause because it does not affect the execution of `getopts`; it is one of the few ''output-only'' variables used by the standard utilities.

Use of colon (`:`) as an option character (in a previous draft) was new behavior and violated the syntax guidelines. Many objectors felt that it did not add enough to `getopts` to warrant mandating the extension to existing practice. The colon is now specified to behave as in the KornShell version of the `getopts` utility; when used as the first character in the *optstring* operand, it disables diagnostics concerning missing option-arguments and unexpected option characters. This replaces the use of the **OPTERR** variable that was specified in an earlier draft.

The formats of the diagnostic messages produced by the `getopts` utility and the *getopt* () function are not fully specified because implementations with superior (''friendlier'') formats objected to the formats used by some historical implementations. It was felt to be important that the information in the messages used be uniform between `getopts` and *getopt* (). Exact duplication of the messages might not be possible, particularly if a utility is built on another system that has a different *getopt* () function, but the messages must have specific information included so that the program name, invalid option character, and type of error can be distinguished by a user.

Only a rare application program will intercept a `getopts` standard error message and want to parse it. Therefore, implementations are free to choose the most usable messages they can devise. The following formats are used by many historical implementations:

> `"%s: illegal option -- %c\n"`, *<program name>* ,
> *<option character>*

> `"%s: option requires an argument -- %c\n"`,
> *<program name>* , *<option character>*

Historical shells with built-in versions of *getopt* () or `getopts` have used different formats, frequently not even indicating the option character found in error.

4.28 `grep` — File pattern searcher

4.28.1 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 ... ]
```

Obsolescent Versions :

```
egrep [ −c | −l ] [−inv] −e pattern_list [file ... ]
egrep [ −c | −l ] [−inv] −f pattern_file [file ... ]
egrep [ −c | −l ] [−inv] pattern_list [file ... ]
fgrep [ −c | −l ] [−invx] −e pattern_list [file ... ]
fgrep [ −c | −l ] [−invx] −f pattern_file [file ... ]
fgrep [ −c | −l ] [−invx] pattern_list [file ... ]
```

4.28.2 Description

The `grep` utility shall search the input files, selecting lines matching one or more patterns; the types of patterns shall be 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 shall consist of one or more patterns separated by <newline>s; the *pattern_file* 's contents shall consist of one or more patterns terminated by <newline>s. By default, an input line shall be selected if any pattern, treated as an entire basic regular expression (BRE) as described in 2.8.3, matches any part of the line; a null BRE shall match every line. By default, each selected input line shall be written to the standard output.

Regular expression matching shall be based on text lines. Since <newline> 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> found in the input.

A command invoking the (obsolescent) `egrep` utility with the `−e` option specified shall be equivalent to the command:

`grep −E [ −c | −l ] [−inv] −e` *pattern_list* [*file* . . .]

A command invoking the `egrep` utility with the `−f` option specified shall be equivalent to the command:

`grep −E [ −c | −l ] [−inv] −f` *pattern_file* [*file* . . .]

A command invoking the `egrep` utility with the *pattern_list* specified shall be equivalent to the command:

`grep −E [ −c | −l ] [−inv]` *pattern_list* [*file* . . .]

A command invoking the (obsolescent) `fgrep` utility with the `−e` option specified shall be equivalent to the command:

`grep −F [ −c | −l ] [−invx] −e` *pattern_list* [*file* . . .]

A command invoking the `fgrep` utility with the `−f` option specified shall be equivalent to the command:

`grep −F [ −c | −l ] [−invx] −f` *pattern_file* [*file* . . .]

A command invoking the `fgrep` utility with the *pattern_list* operand specified shall be equivalent to the command:

`grep −F [ −c | −l ] [−invx]` *pattern_list* [*file* . . .]

4.28.3 Options

The `grep` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−E` Match using extended regular expressions. Treat each pattern specified as an ERE, as described in 2.8.4. If any entire ERE pattern matches an input line, the line shall be matched. A null ERE shall match 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 shall be matched. A null string shall match 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* shall be separated by a `<newline>`. A null pattern can be specified by two adjacent `<newline>`s in *pattern_list*; in the obsolescent forms, adjacent `<newline>`s in *pattern_list* produce undefined results. Unless the `−E` or `−F` option is also specified, each pattern shall be treated as a BRE, as described in 2.8.3. In the nonobsolescent forms, multiple `−e` and `−f` options shall be accepted by the `grep` utility. All of the specified patterns shall be 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* shall be terminated by a `<newline>`. A null pattern can be specified by an empty line in *pattern_file*. Unless the `−E` or `−F` option is also specified, each pattern shall be treated as a BRE, as described in 2.8.3.

`−i` Perform pattern matching in searches without regard to case. See 2.8.2.

`−l` (The letter ell.) Write only the names of files containing selected lines to standard output. Pathnames shall be written once per file searched. If the standard input is searched, a pathname of `"(standard input)"` shall 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 shall 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 nonexistent or unreadable files. Other error messages shall not be suppressed.

`−v` Select lines not matching any of the specified patterns. If the `−v` option is not specified, selected lines shall 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.

4.28.4 Operands

The following operands shall be supported by the implementation:

pattern Specify one or more patterns to be used during the search for input. This operand shall be treated as if it were specified as `−e` *pattern_list* (see 4.28.3).

file A pathname of a file to be searched for the pattern(s). If no *file* operands are specified, the standard input shall be used.

4.28.5 External Influences

4.28.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.28.5.2 Input Files

The input files shall be text files.

4.28.5.3 Environment Variables

The following environment variables shall affect the execution of `grep`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes within regular expressions.

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

4.28.5.4 Asynchronous Events

Default.

4.28.6 External Effects

4.28.6.1 Standard Output

If the `-l` option is in effect, and the `-q` option is not, a single output line shall 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 shall prefix each output line by:

`"%s:"`, *file*

The remainder of each output line shall depend on the other options specified:

— If the `-c` option is in effect, the remainder of each output line shall contain:

`"%d\n"`, *<count>*

— Otherwise, if `-c` is not in effect and the `-n` option is in effect, the following shall be written to standard output:

`"%d:"`, *<line number>*

— Finally, the following shall be written to standard output:

`"%s"`, *<selected-line contents>*

4.28.6.2 Standard Error

Used only for diagnostic messages.

4.28.6.3 Output Files

None.

4.28.7 Extended Description

None.

4.28.8 Exit Status

The `grep` utility shall exit with one of the following values:

- 0 One or more lines were selected.
- 1 No lines were selected.
- >1 An error occurred.

4.28.9 Consequences of Errors

If the `-q` option is specified, the exit status shall be zero if an input line is selected, even if an error was detected. Otherwise, default actions shall be performed.

4.28.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

This `grep` has been enhanced in an upward-compatible way to provide the exact functionality of the historical `egrep` and `fgrep` commands as well. It was the clear intention of the working group to consolidate the three `grep`s into a single command.

The old `egrep` and `fgrep` commands are likely to be supported for many years to 1
come as implementation extensions, allowing existing applications to operate unmodified.

To find all uses of the word `Posix` (in any case) in the file `text.mm`, and write with line numbers:

```
grep -i -n posix text.mm
```

To find all empty lines in the standard input: 2

```
grep ^$
    or
grep -v .
```

Both of the following commands print all lines containing strings `abc` or `def` or both:

```
grep -E 'abc
def'
```

```
grep -F 'abc
def'
```

Both of the following commands print all lines matching exactly `abc` or `def`:

```
grep -E '^abc$
^def$'
```

```
grep -F -x 'abc
def'
```

History of Decisions Made

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.

Earlier drafts did not show that the `−c`, `−l`, and `−q` options were mutually exclusive. This has been fixed to more closely align with historical practice and documentation.

Historical implementations usually silently ignored all but one of multiply specified `−e` and `−f` options, but were not consistent as to which specification was actually used.

POSIX.2 requires that the nonobsolescent forms accept multiple `−e` and `−f` options and use all of the patterns specified 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 `−b` option was removed from the Options subclause, since block numbers are implementation dependent.

The System V restriction on using `−` to mean standard input was lifted.

A definition of action taken when given a null RE or ERE is specified. This is an error condition in some historical implementations.

The `−l` option previously indicated that its use was undefined when no files were explicitly named. This behavior was historical and placed an unnecessary restriction on future implementations. It has been removed.

The `−q` option was added at the suggestion of members of the balloting group as 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).

The historical BSD `grep −s` option practice is easily duplicated by redirecting standard output to `/dev/null`. The `−s` option required here is from System V.

The `−x` option, historically available only with `fgrep`, is available here for all of the nonobsolescent versions.

4.29 head — Copy the first part of files

4.29.1 Synopsis

`head` [`−n` *number*] [*file* ...]

Obsolescent version:

`head` [`−`*number*] [*file* ...]

4.29.2 Description

The `head` utility shall copy its input files to the standard output, ending the output for each file at a designated point.

Copying shall end at the point in each input file indicated by the `−n` *number* option (or the obsolescent version's `−`*number* argument). The option-argument *number* shall be counted in units of lines.

4.29.3 Options

The `head` utility shall conform to the utility argument syntax guidelines described in standard described in 2.10.2, except that the obsolescent version accepts multicharacter numeric options.

The following option shall be supported by the implementation in the nonobsolescent version:

`−n` *number* The first *number* lines of each input file shall be copied to standard output. The *number* option argument shall be a positive decimal integer.

If no options are specified, `head` shall act as if `-n 10` had been specified.

In the obsolescent version, the following option shall be supported by the implementation:

`−`*number* The *number* argument is a positive decimal integer with the same effect as the `−n`*number* option in the nonobsolescent version.

4.29.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

4.29.5 External Influences

4.29.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.29.5.2 Input Files

Input files shall be text files, but the line length shall not be restricted to {LINE_MAX} bytes.

4.29.5.3 Environment Variables

The following environment variables shall affect the execution of `head`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.29.5.4 Asynchronous Events

Default.

4.29.6 External Effects

4.29.6.1 Standard Output

The standard output shall contain designated portions of the input file(s).

If multiple *file* operands are specified, `head` shall precede the output for each with the header:

`"\n==> %s <==\n"`, <*pathname*>

except that the first header written shall not include the initial `<newline>`.

4.29.6.2 Standard Error

Used only for diagnostic messages.

4.29.6.3 Output Files

None.

4.29.7 Extended Description

None.

4.29.8 Exit Status

The `head` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.29.9 Consequences of Errors

Default.

4.29.10 Rationale. *(This subclause is not a part of P1003.2)*

Usage, Examples

The nonobsolescent version of `head` was created to allow conformance to the Utility Syntax Guidelines. The `-n` option was added to this new interface so that `head` and `tail` would be more logically related.

To write the first ten lines of all files (except those with a leading period) in the directory:

```
head *
```

History of Decisions Made

The `head` utility was not in early drafts. It was felt that `head`, and its frequent companion, `tail`, were useful mostly to interactive users, and not application programs. However, balloting input suggested that these utilities actually do find significant use in scripts, such as to write out portions of log files. Although it is possible to simulate `head` with `sed 10q` for a single file, the working group decided that the popularity of `head` on historical BSD systems warranted its inclusion alongside `tail`.

An earlier draft had the synopsis line:

head [−c | −l] [−n *number*] [*file* . . .]

This was changed to the current form based on comments and objections noting that `−c` has not been provided by historical versions of `head` and other utilities in POSIX.2 provide similar functionality. Also, `−l` was changed to `−n` to match a similar change in `tail`.

4.30 `id` — Return user identity

4.30.1 Synopsis

id [*user*]

id −G [−n] [*user*]

id −g [−nr] [*user*]

id −u [−nr] [*user*]

4.30.2 Description

If no *user* operand is provided, the `id` utility shall 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 shall be written. If multiple groups are supported by the underlying system (see the description of {NGROUPS_MAX} in POSIX.1 {8}), the supplementary group affiliations of the invoking process also shall be written.

If a *user* operand is provided and the process has the appropriate privileges, the user and group IDs of the selected user shall be written. In this case, effective IDs shall be assumed to be identical to real IDs. If the selected user has more than one allowable group membership listed in the group database (see POSIX.1 {8} section 9.1), these shall be written in the same manner as the supplementary groups described in the preceding paragraph.

4.30.3 Options

The `id` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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>` 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"`.

4.30.4 Operands

The following operand shall be supported by the implementation:

user The login name for which information is to be written.

4.30.5 External Influences

4.30.5.1 Standard Input

None.

4.30.5.2 Input Files

None.

4.30.5.3 Environment Variables

The following environment variables shall affect the execution of `id`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.30.5.4 Asynchronous Events

Default.

4.30.6 External Effects

4.30.6.1 Standard Output

The following formats shall 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 shall 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 shall 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 1
to belong to multiple groups, the first shall be added directly before the `<newline>` 1
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 1
supplementary/multiple group IDs cannot be mapped by the system into printable 1
user or group names, the corresponding (`%s`) and name argument shall be omitted from the corresponding format string.

When any of the options are specified, the output format shall be as described under 4.30.3.

4.30.6.2 Standard Error

Used only for diagnostic messages.

4.30.6.3 Output Files

None.

4.30.7 Extended Description

None.

4.30.8 Exit Status

The `id` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.30.9 Consequences of Errors

Default.

4.30.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The functionality provided by the 4BSD `groups` utility can be simulated using:

`id -Gn [`*user*`]`

Note that 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` (see 4.25) before postprocessing the output of `id`.

History of Decisions Made

The 4BSD command `groups` was considered, but was not used as it did not provide the functionality of the `id` utility of the *SVID*. Also, it was thought that it would be easier to modify `id` to provide the additional functionality necessary to systems with multiple groups than to invent another command.

The options `−u`, `−g`, `−n`, and `−r` were added to ease the use of `id` with shell commands substitution. Without these options it is necessary to use some preprocessor such as `sed` to select the desired piece of information. Since output such as that produced by `id -u -n` is wanted frequently, it seemed desirable to add the options.

4.31 `join` — Relational database operator

4.31.1 Synopsis

`join` [`−a` *file_number* | `−v` *file_number*] [`−e` *string*] [`−o` *list*] [`−t` *char*]
    [`−1` *field*] [`−2` *field*] *file1 file2*

Obsolescent version:

`join` [`−a` *file_number*] [`−e` *string*] [`−j` *field*] [`−j1` *field*] [`−j2` *field*]
    [`−o` *list* ...] [`−t` *char*] *file1 file2*

4.31.2 Description

The `join` utility shall perform an ‘‘equality join’’ on the files *file1* and *file2*. The joined files shall be written to the standard output.

The ‘‘join field’’ is a field in each file on which the files are compared. There shall be one line in the output for each pair of lines in *file1* and *file2* that have identical join fields. The output line by default shall 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` (see 4.58) on the fields on which they are to be joined, by default the first in each line. All selected output shall be written in the same collating sequence.

The default input field separators shall be `<blank>`s. In this case, multiple separators shall count as one field separator, and leading separators shall be ignored. The default output field separator shall be a `<space>`.

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.

4.31.3 Options

The `join` utility shall conform to the utility argument syntax guidelines described in 2.10.2. The obsolescent version does not follow the utility argument syntax guidelines: the `−j1` and `−j2` options are multicharacter options and the `−o` option takes multiple arguments.

The following options shall be supported by the implementation:

`−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 shall be output.

`−e` *string*
: Replace empty output fields by string *string* .

`−j` *field*
: (Obsolescent.) Equivalent to: `−1` *field* `−2` *field*

`−j1` *field*
: (Obsolescent.) Equivalent to: `−1` *field*

`−j2` *field*
: (Obsolescent.) Equivalent to: `−2` *field*

`−o` *list*
: Construct the output line to comprise the fields specified in *list* , each element of which has the form *file_number* `.`*field* , where *file_number* is a file number and *field* is a decimal integer field number. The elements of *list* are either comma- or `<blank>`-separated, as specified in Guideline 8 in 2.10.2. The fields specified by *list* shall be written for all selected output lines. Fields selected by *list* that do not appear in the input shall be treated as empty output fields. (See the `−e` option.) The join field shall not be written unless specifically requested. The *list* shall be a single command line argument. However, as an obsolescent feature, the argument *list* can be multiple arguments on the command line. If this is the case, and if the `−o` option is the last option before *file1* , and if *file1* is of the form *string* .*string* , the results are undefined.

`−t` *char* Use character *char* as a separator, for both input and output. Every appearance of *char* in a line shall 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 shall 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.

4.31.4 Operands

The following operands shall be supported by the implementation:

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.

4.31.5 External Influences

4.31.5.1 Standard Input

The standard input shall be used only if the *file1* or *file2* operand is −. See Input Files.

4.31.5.2 Input Files

The input files shall be text files.

4.31.5.3 Environment Variables

The following environment variables shall affect the execution of `join`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the collating sequence `join` expects to have been used when the input files were sorted.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.31.5.4 Asynchronous Events

Default.

4.31.6 External Effects

4.31.6.1 Standard Output

The `join` utility output shall be a concatenation of selected character fields. When the `−o` option is not specified, the output shall be:

`"%s%s%s\n"`, *<join field>*, *<other file1 fields>*, *<other file2 fields>*

If the join field is not the first field in either file, the *<other file fields>* are:

<fields preceding join field>, *<fields following join field>*

When the `−o` option is specified, the output format shall 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) shall be written with its trailing separator character. If the separator is the default (`<blank>`s), a single `<space>` character shall be written after each field (except the last).

4.31.6.2 Standard Error

Used only for diagnostic messages.

4.31.6.3 Output Files

None.

4.31.7 Extended Description

None.

4.31.8 Exit Status

The `join` utility shall exit with one of the following values:

0 All input files were output successfully.

>0 An error occurred.

4.31.9 Consequences of Errors

Default.

4.31.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Pathnames consisting of numeric digits should not be specified directly following the `−o` list.

The developers of the standard believed that `join` should operate as documented in the *SVID* and BSD, not as historically implemented. Historical implementations do not behave as documented in these areas:

(1) Most implementations of `join` require using the `−o` option when using the `−e` option.

(2) Most implementations do not parse the `−o` option as documented, and parse the elements as separate *argv* items, until the item is not of the form *file_number* `.`*field*. This behavior is permitted as an obsolescent usage of the utility. To ensure maximum portability, *file1* should not be of the form *string*.*string*. A suitable alternative to guarantee portability would be to put the `−−` flag before any *file1* operand.

The obsolescent `−j`, `−j1`, and `−j2` options have been described to show how they have been used in historical implementations. Earlier drafts showed `−j` *file_number field*, but a space was never allowed before the *file_number* and two option arguments were never intended.

History of Decisions Made

The ability to specify *file2* as `−` is not historical practice; it was added for completeness.

As a result of a balloting comment, the `−v` option was added to the nonobsolescent version. This option was felt necessary because it permitted the writing of *only* those lines that do not match on the join field, as opposed to the `−a` option, which prints both lines that do and do not match. This additional facility is parallel with the `−v` option of `grep`.

4.32 `kill` — Terminate or signal processes

4.32.1 Synopsis

`kill −s` *signal_name pid* . . .

`kill −l` [*exit_status*]

Obsolescent Versions :

`kill` [−*signal_name*] *pid* . . .

`kill` [−*signal_number*] *pid* . . .

4.32.2 Description

The `kill` utility shall send a signal to the process(es) specified by each *pid* operand.

For each *pid* operand, the `kill` utility shall perform actions equivalent to the POSIX.1 {8} *kill*() function called with the following arguments:

(1) The value of the *pid* operand shall 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.

4.32.3 Options

The `kill` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that in the obsolescent form, the −*signal_number* and −*signal_name* options are usually more than a single character.

The following options shall be supported by the implementation:

`−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 3.5.2 and `wait` in 4.70) corresponding to a process that was terminated by a signal, the *signal_name* corresponding to the signal that terminated the process shall 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 POSIX.1 {8}-defined symbolic constant name without the `SIG` prefix) corresponding to that signal shall be written. Otherwise, the results are unspecified.

−s *signal_name*
: Specify the signal to send, using one of the symbolic names defined for Required Signals or Job Control Signals in POSIX.1 {8} 3.3.1.1. Values of *signal_name* shall be recognized in a case-independent fashion, without the SIG prefix. In addition, the symbolic name 0 shall be recognized, representing the signal value zero. The corresponding signal shall be sent instead of SIGTERM.

−*signal_name*
: (Obsolescent.) Equivalent to −s *signal_name.*

−*signal_number*
: (Obsolescent.) Specify a nonnegative 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.

In the obsolescent versions, if the first argument is a negative integer, it shall be interpreted as a −*signal_number* option, not as a negative *pid* operand specifying a process group.

4.32.4 Operands

The following operands shall be supported by the implementation:

pid
: A decimal integer specifying a process or process group to be signaled. The process(es) selected by positive, negative, and zero values of the *pid* operand shall be as described for POSIX.1 {8} *kill*() function. If the first *pid* operand is negative, it should be preceded by -- to keep it from being interpreted as an option.

exit_status A decimal integer specifying a signal number or the exit status of a process terminated by a signal.

4.32.5 External Influences

4.32.5.1 Standard Input

None.

4.32.5.2 Input Files

None.

4.32.5.3 Environment Variables

The following environment variables shall affect the execution of `kill`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.32.5.4 Asynchronous Events

Default.

4.32.6 External Effects

4.32.6.1 Standard Output

When the `-l` option is not specified, the standard output shall not be used.

When the `-l` option is specified, the symbolic name of each signal shall be written in the following format:

`"%s%c"`, *<signal_name>* , *<separator>*

where the *<signal_name>* is in uppercase, without the `SIG` prefix, and the

<separator> shall be either a `<newline>` or a `<space>`. For the last signal written, *<separator>* shall be a `<newline>`.

When both the `−l` option and *exit_status* operand are specified, the symbolic name of the corresponding signal shall be written in the following format:

"%s\n", *<signal_name>*

4.32.6.2 Standard Error

Used only for diagnostic messages.

4.32.6.3 Output Files

None.

4.32.7 Extended Description

None.

4.32.8 Exit Status

The `kill` utility shall exit with one of the following values:

- 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.

4.32.9 Consequences of Errors

Default.

4.32.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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.

POSIX.1 {8} and POSIX.2 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
```

History of Decisions Made

The signal name extension was based on a desire to avoid limiting the `kill` utility to implementation-dependent values.

The `−l` option originated from the C-shell, and is also implemented in the KornShell. The C-shell output can consist of multiple output lines, because the signal names do not always fit on a single line on some terminal screens. The KornShell output also included the implementation-specific signal numbers, and was felt by the working group to be too difficult for scripts to parse conveniently. The specified output format is intended not only to accommodate the historical C-shell output, but also to permit an entirely vertical or entirely horizontal listing on systems for which this is appropriate.

An earlier draft invented the name SIGNULL as a *signal_name* for signal 0 (used by POSIX.1 {8} to test for the existence of a process without sending it a signal). Since the *signal_name* `"0"` can be used in this case unambiguously, SIGNULL has been removed.

An earlier draft also required symbolic *signal_name* s to be recognized with or without the `SIG` prefix. Historical versions of `kill` have not written the `SIG` prefix for the `−l` option and have not recognized the `SIG` prefix on *signal_name* s. Since neither application portability nor ease of use would be improved by requiring this extension, it is no longer required.

POSIX.2 contains no utility that browses for process IDs. Values for *pid* are available via the `!` and `$` parameters of the shell command language (see 3.5.2).

The use of numeric signal values was the subject of a long debate in the Working Group. During balloting, it was determined that their use should be declared obsolescent, but retained to provide backward compatibility to existing applications.

Existing implementations of `kill` permit negative *pid* operands representing process groups, but this was often unclearly documented. The assumption that an initial negative number argument specifies a signal number (rather than a process group) is the existing behavior, and was retained. 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
```

The `-s` option was added in response to international interest in providing some form of `kill` that meets the Utility Syntax Guidelines.

Some implementations provide `kill` only as a shell built-in utility and use that status to support the extension of killing background asynchronous lists (those started with &), by the use of job identifiers. For example,

```
kill %1
```

would `kill` the first asynchronous list in the background. This standard does not require (but permits) such an extension, because other related job-control features are not provided by the shell, and because these facilities are not ordinarily usable in portable shell applications. This notation is expected to be introduced by the UPE.

4.33 `ln` — Link files

4.33.1 Synopsis

`ln [-f]` *source_file target_file*

`ln [-f]` *source_file* . . . *target_dir*

4.33.2 Description

In the first synopsis form, the `ln` utility shall 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 shall be assumed when the final operand does not name an existing directory; if more than two operands are 1
specified and the final is not an existing directory, an error shall result. 1

In the second synopsis form, the `ln` utility shall 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 POSIX.1 {8}, the behavior is implementation defined.

The corresponding destination path for each *source_file* shall be the concatenation of the target directory pathname, a slash character, and the last pathname component of the *source_file*. The second synopsis form shall 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` shall 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 shall be performed equivalent to the POSIX.1 {8} *unlink* () function, called using *destination* as the *path* argument. If this fails for any reason, `ln` shall 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 shall be performed equivalent to the POSIX.1 {8} *link* () function using *source_file* as the *path1* argument, and the *destination* path as the *path2* argument.

4.33.3 Options

The `ln` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−f` Force existing *destination* pathnames to be removed to allow the link.

4.33.4 Operands

The following operands shall be supported by the implementation:

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 defined.

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.

4.33.5 External Influences

4.33.5.1 Standard Input

None.

4.33.5.2 Input Files

None.

4.33.5.3 Environment Variables

The following environment variables shall affect the execution of `ln`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.33.5.4 Asynchronous Events

Default.

4.33.6 External Effects

4.33.6.1 Standard Output

None.

4.33.6.2 Standard Error

Used only for diagnostic messages.

4.33.6.3 Output Files

None.

4.33.7 Extended Description

None.

4.33.8 Exit Status

The `ln` utility shall exit with one of the following values:

0	All the specified files were linked successfully.
>0	An error occurred.

4.33.9 Consequences of Errors

Default.

4.33.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

Some historic versions of `ln` (including the one specified by the *SVID*) unlink the destination file, if it exists, by default. If the mode does not permit writing, these versions will prompt for confirmation before attempting the unlink. In these versions the `−f` option causes `ln` to not attempt to prompt for confirmation.

This allows `ln` to succeed in creating links when the target file already exists, even if the file itself is not writable (although the directory must be). Previous versions of this draft specified this functionality.

This draft does not allow the `ln` utility to unlink existing destination paths by default for the following reasons:

— The `ln` utility has traditionally been used to provide locking for shell applications, a usage that is incompatible with `ln` unlinking the destination path by default. There was no corresponding technical advantage to adding this functionality.

— This functionality gave `ln` the ability to destroy the link structure of files, which changes the historical behavior of `ln`.

— This functionality is easily replicated with a combination of `rm` and `ln`.

— It is not historical practice in many systems; BSD and BSD-derived systems do not support this behavior. Unfortunately, whichever behavior is selected can cause scripts written expecting the other behavior to fail.

— It is preferable that `ln` perform in the same manner as the *link* () function, which does not permit the target to already exist.

This standard retains the `−f` option to provide support for shell scripts depending on the *SVID* semantics. It seems likely that shell scripts would not be written to handle prompting by `ln`, and would therefore have specified the `−f` option.

It should also be noted that `−f` is an undocumented feature of many historical versions of the `ln` utility, allowing linking to directories. These versions will require modification.

Previous drafts of this standard also required an `−i` option, which behaved like the `−i` options in `cp` and `mv`, prompting for confirmation before unlinking existing files. This was not historical practice for the `ln` utility and has been deleted from this version.

Although symbolic links are not part of the standard, the `−s` option should be used only for the traditional purpose of creating symbolic links.

4.34 `locale` — Get locale-specific information

4.34.1 Synopsis

```
locale [ −a | −m ]
locale [−ck] name ...
```

4.34.2 Description

The `locale` utility shall write information about the current locale environment, or all public locales, to the standard output. For the purposes of this clause, a *public locale* is one provided by the implementation that is accessible to the application.

When `locale` is invoked without any arguments, it shall summarize the current locale environment for each locale category as determined by the settings of the environment variables defined in 2.5.

When invoked with operands, it shall write values that have been assigned to the keywords in the locale categories, as follows:

- Specifying a keyword name shall select the named keyword and the category containing that keyword.
- Specifying a category name shall select the named category and all keywords in that category.

4.34.3 Options

The `locale` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−a` Write information about all available public locales. The available locales shall include `POSIX`, representing the POSIX Locale. The manner in which the implementation determines what other locales are available is implementation defined.

`−c` Write the names of selected locale categories; see 4.34.6.1.

`−k` Write the names and values of selected keywords. The implementation may omit values for some keywords; see 4.34.4.

`−m` Write names of available charmaps; see 2.4.1. 1

4.34.4 Operands

The following operand shall be supported by the implementation:

name The name of a locale category as defined in 2.5, the name of a keyword in a locale category, or the reserved name `charmap`. The named category or keyword shall 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 defined whether any keyword values are written for the categories LC_CTYPE and LC_COLLATE.

4.34.5 External Influences

4.34.5.1 Standard Input

None.

4.34.5.2 Input Files

None.

4.34.5.3 Environment Variables

The following environment variables shall affect the execution of `locale`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

The **LANG** and **LC_*** environment variables shall specify the current locale environment to be written out; they shall be used if the `-a` option is not specified.

4.34.5.4 Asynchronous Events

Default.

4.34.6 External Effects

4.34.6.1 Standard Output

If `locale` is invoked without any options or operands, the names and values of the **LANG** and **LC_*** environment variables described in this standard shall 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** shall be written using this format:

`"%s=%s\n"`, *<variable_name>* , *<value>*

The names of those **LC_*** variables associated with locale categories defined in this standard that are not set in the environment or are overridden by **LC_ALL** shall be written in the following format:

`"%s=\"%s\"\n"`, *<variable_name>* , *<implied value>*

The *<implied value>* shall be 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 2.6.

The *<value>* and *<implied value>* shown above shall be properly quoted for possi- 1
ble later re-entry to the shell. The *<value>* shall not be quoted using double- 1
quotes (so that it can be distinguished by the user from the *<implied value>* case, 1
which always requires double-quotes). 1

The **LC_ALL** variable shall be written last, using the first format shown above. If it is not set, it shall be written as: 1

`"LC_ALL=\n"`

If any arguments are specified:

(1) If the `−a` option is specified, the names of all the public locales shall be written, each in the following format:

`"%s\n"`, *<locale name>*

(2) If the `−c` option is specified, the name(s) of all selected categories shall 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 shall precede the keyword output for that category.

If the `−c` option is not specified, the names of the categories shall not be written; only the keywords, as selected by the *name* operand, shall be written. 2

(3) If the `−k` option is specified, the name(s) and value(s) of selected keywords shall be written. If a value is nonnumeric, it shall 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 shall be written, with the word `charmap` as *<keyword name>* .

If a value is numeric, it shall 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 2.5.2.

Compound keyword values (list entries) shall be separated in the output by semicolons. When included in keyword values, the semicolon, the double-quote, the backslash, and any control character shall be preceded (escaped) with the escape character.

(4) If the `−k` option is not specified, selected keyword values shall 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 shall be written.

(5) If the `−m` option is specified, then a list of all available charmaps shall 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.

4.34.6.2 Standard Error

Used only for diagnostic messages.

4.34.6.3 Output Files

None.

4.34.7 Extended Description

None.

4.34.8 Exit Status

The `locale` utility shall exit with one of the following values:

- 0 All the requested information was found and output successfully.
- >0 An error occurred.

4.34.9 Consequences of Errors

Default.

4.34.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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 standard.

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
        nonaffirmative processing goes here
fi
```

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 unrecognized value, the actual locales assumed (if any) are implementation defined as described in 2.6.

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, e.g., which character classes are available).

History of Decisions Made

This command was added in Draft 9 to resolve objections to the lack of a way for applications to determine what locales are available, a way to examine the contents of existing public locales, a way to retrieve specific locale items, and a way to recognize affirmative and negative responses in an international environment.

In Draft 10 it was cut back considerably in answer to balloting objections about its complexity and requirement of features not useful for application programs. The format for the no-arguments case was expanded to show the implied values of the categories as an aid to the novice user; the output was of little more value than that from `env`.

Based on the questionable value in a shell script of getting an entire array of characters back, and the problem of returning a collation description that makes sense, short of a complete `localedef` source, the output from requests for categories LC_CTYPE and LC_COLLATE has been made implementation defined.

The `-m` option has been added to allow applications to query for the existence of charmaps. The output is a list of the charmaps (implementation-supplied and user-supplied, if any) on the system.

The `-c` option was included for readability when more than one category is 2
selected (e.g., via more than one keyword name or via a category name). It is 2
valid both with and without the `-k` option. 2

The `charmap` keyword, which returns the name of the charmap (if any) that was used when the current locale was created, was introduced to allow applications needing the information to retrieve it.

4.35 `localedef` — Define locale environment

4.35.1 Synopsis

`localedef [-c][-f` *charmap* `][-i` *sourcefile* `]` *name*

4.35.2 Description

The `localedef` utility shall convert source definitions for locale categories into a format usable by the functions and utilities whose operational behavior is determined by the setting of the locale environment variables defined in 2.5. It is implementation defined whether users shall have the capability to create new locales, in addition to those supplied by the implementation. If the symbolic constant {POSIX 2_LOCALEDEF } is defined, then the system supports the creation of new locales. In a system not supporting this capability, the `localedef` utility shall terminate with an exit code of 3.

The utility shall read 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 shall support 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 shall be identified by the corresponding environment variable name and terminated by an `END` *category-name* statement. The following categories shall be supported. In addition, the input may contain source for implementation-defined 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 nonmonetary 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.

4.35.3 Options

The `localedef` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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 2.4.1. This option shall 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-defined default character mapping file shall be used. 2

`−i` *inputfile* The pathname of a file containing the source definitions. If this option is not present, source definitions shall be read from standard input. The format of the *inputfile* is described in 2.5.2.

4.35.4 Operands

The following operand shall be supported by the implementation:

name Identifies the locale. See 2.5 for a description of the use of this name. If the name contains one or more slash characters, *name* shall be interpreted as a pathname where the created locale definition(s) shall be stored. If *name* does not contain any slash characters, the interpretation of the name is implementation defined and the locale shall be public. This capability may be restricted to users with appropriate privileges.

4.35.5 External Influences

4.35.5.1 Standard Input

Unless the `-i` option is specified, the standard input shall be a text file containing
one or more locale category source definitions, as described in 2.5.2. When lines 1
are continued using the escape character mechanism, there is no limit to the 1
length of the accumulated continued line. 1

4.35.5.2 Input Files

The character set mapping file specified as the *charmap* option-argument is described under 2.4.1. If a locale category source definition contains a `copy` statement, as defined in 2.5.2, and the `copy` statement names a valid, existing locale, then `localedef` shall behave as if the source definition had contained a valid category source definition for the named locale.

4.35.5.3 Environment Variables

The following environment variables shall affect the execution of `localedef`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**. and **LC_*** variables as described in 2.6.

LC_COLLATE (This variable shall have no affect on `localedef`; the POSIX Locale shall be used for this category.)

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of argument data as characters (e.g., single- versus multibyte characters). This variable shall have no affect on the processing of `localedef` input data; the POSIX Locale shall be used for this purpose, regardless of the value of this variable.

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

4.35.5.4 Asynchronous Events

Default.

4.35.6 External Effects

4.35.6.1 Standard Output

The utility shall report all categories successfully processed, in an unspecified format.

4.35.6.2 Standard Error

Used only for diagnostic messages.

4.35.6.3 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.

4.35.7 Extended Description

None.

4.35.8 Exit Status

The `localedef` utility shall exit with one of the following values:

0 No errors occurred and the locale(s) were successfully created.

1 Warnings occurred and the locale(s) 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.

4.35.9 Consequences of Errors

If an error is detected, no permanent output shall be created.

If warnings occur, permanent output shall be created if the `-c` option was specified. The following conditions shall 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 shall 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 1
the source. 1

Other implementation-defined conditions may also cause warnings.

4.35.10 Rationale. *(This subclause is not a part of P1003.2)*

Usage, Examples

The output produced by the `localedef` utility is implementation defined. The *name* operand is used to identify the specific locale. (As a consequence, although several categories can be processed in one execution, only categories belonging to the same locale can be processed.)

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 code set). To aid portability, all *charmap* definitions shall use the same symbolic names for the portable character set. As explained in 2.4.1, it is implementation defined 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.

History of Decisions Made

This description is based on work performed in the UniForum Technical Committee Subcommittee on Internationalization.

The `localedef` utility is provided as a standard, portable interface for implementations that allow users to create new locales, in addition to implementation-supplied ones.

The ability to create new locales and categories, already available on many commercially available implementations of POSIX compliant systems, provides the means by which application providers can develop portable applications which use standard interfaces to adjust the behavior of the application to language and culture differences.

4.36 `logger` — Log messages

4.36.1 Synopsis

`logger` *string* ...

4.36.2 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.

4.36.3 Options

None.

4.36.4 Operands

The following operands shall be supported by the implementation:

string One of the string arguments whose contents are concatenated together, in the order specified, separated by single `<space>`s.

4.36.5 External Influences

4.36.5.1 Standard Input

None.

4.36.5.2 Input Files

None.

4.36.5.3 Environment Variables

The following environment variables shall affect the execution of `logger`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

LC_MESSAGES This variable shall determine the language in which diagnostic messages should be written.

4.36.5.4 Asynchronous Events

Default.

4.36.6 External Effects

4.36.6.1 Standard Output

None.

4.36.6.2 Standard Error

Used only for diagnostic messages.

4.36.6.3 Output Files

Unspecified.

4.36.7 Extended Description

None.

4.36.8 Exit Status

The `logger` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.36.9 Consequences of Errors

Default.

4.36.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

This utility allows logging of information for later use by a system administrator or programmer in determining why noninteractive utilities have failed. POSIX.2 makes no requirements for the locations of the saved message, their format, or retention period. It also provides no method for a portable application to read messages, once written. (It is expected that the POSIX.7 System Administration standard will have something to say about that.)

The purpose of this utility might best be illustrated by an example. A batch application, running noninteractively, tries to read a configuration file and fails; it may attempt to notify the system administrator with:

```
logger myname: unable to read file foo. [time stamp]
```

The text with **LC_MESSAGES** about diagnostic messages means diagnostics from `logger` to the user or application, not diagnostic messages that the user is sending to the system administrator.

History of Decisions Made

Multiple *string* arguments were allowed, similar to `echo`, for ease of use.

In Draft 9, the `posixlog` utility was renamed `logger` to match its BSD forebear, with which it is (downward) compatible.

The working group believed strongly that some method of alerting administrators to errors was necessary. The obvious example is a batch utility, running noninteractively, that is unable to read its configuration files, or that is unable to create or write its results file. However, the working group did not wish to define the format or delivery mechanisms as they have historically been (and will probably continue to be) very system specific, as well as involving functionality clearly outside of the scope of this standard.

Like the utilities `mailx` and `lp`, `logger` is admittedly difficult to test. This was not deemed sufficient justification to exclude these utilities from the standard. It is also arguable that they are, in fact, testable, but that the tests themselves are not portable.

4.37 `logname` — Return user's login name

4.37.1 Synopsis

```
logname
```

4.37.2 Description

The `logname` utility shall write the user's login name to standard output. The login name shall be the string that would be returned by the POSIX.1 {8} *getlogin* () function. Under the conditions where the *getlogin* () function would fail, the `logname` utility shall write a diagnostic message to standard error and exit with a nonzero exit status.

4.37.3 Options

None.

4.37.4 Operands

None.

4.37.5 External Influences

4.37.5.1 Standard Input

None.

4.37.5.2 Input Files

None.

4.37.5.3 Environment Variables

The following environment variables shall affect the execution of `logname`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

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

4.37.5.4 Asynchronous Events

Default.

4.37.6 External Effects

4.37.6.1 Standard Output

The `logname` utility output shall be a single line consisting of the user's login name:

`"%s\n"`, <*login name*>

4.37.6.2 Standard Error

Used only for diagnostic messages.

4.37.6.3 Output Files

None.

4.37.7 Extended Description

None.

4.37.8 Exit Status

The `logname` utility shall exit with one of the following values:

0	Successful completion.
>0	An error occurred.

4.37.9 Consequences of Errors

Default.

4.37.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `logname` utility explicitly ignores the **LOGNAME** environment variable because environment changes could produce erroneous results.

History of Decisions Made

The `passwd` file is not listed as required, because the implementation may have other means of mapping login names.

4.38 `lp` — Send files to a printer

4.38.1 Synopsis

`lp [−c] [−d` *dest* `] [−n` *copies* `] [`*file* ... `]`

4.38.2 Description

The `lp` utility shall copy the input files to an output device in an unspecified manner. The default output destination should be to a hardcopy device, such as a printer or microfilm recorder, that produces nonvolatile, human-readable documents. If such a device is not available to the application, or if the system provides no such device, the `lp` utility shall exit with a nonzero 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 shall guarantee exclusive access to the device.

4.38.3 Options

The `lp` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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.

`−d` *dest* Specify a string that names the output device or destination. 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 shall take precedence over **LPDEST** , which in turn shall take precedence over **PRINTER** . Results are undefined when *dest* contains a value that is not a valid device or destination name.

`-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 shall be output as an integral whole, not interleaved with portions of other files.

4.38.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a file to be output. If no *file* operands are specified, or if a *file* operand is `-`, the standard input shall 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`.

4.38.5 External Influences

4.38.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is `-`. See Input Files.

4.38.5.2 Input Files

The input files shall be text files.

4.38.5.3 Environment Variables

The following environment variables shall affect the execution of `lp`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

LPDEST This variable shall be interpreted as a string that names the output device or destination. If the **LPDEST** environment variable is not set, the **PRINTER** environment variable shall be used. The `-d` *dest* option shall take 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.

PRINTER This variable shall be interpreted as a string that names 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 shall take 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.

4.38.5.4 Asynchronous Events

Default.

4.38.6 External Effects

4.38.6.1 Standard Output

A message concerning the identification or status of the print request may be 2
written, in an unspecified format. 2

4.38.6.2 Standard Error

Used only for diagnostic messages.

4.38.6.3 Output Files

None.

4.38.7 Extended Description

None.

4.38.8 Exit Status

The `lp` utility shall exit with one of the following values:

0 All input files were processed successfully.

>0 No output device was available, or an error occurred.

4.38.9 Consequences of Errors

Default.

4.38.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Since the default destination, device type, queueing mechanisms, and acceptable forms of input are all unspecified, usage guidelines for what a portable application can do are as follows:

(1) Use the command in a pipeline, or with `−c`, so that there are no permission problems and the files can be safely deleted or modified.

(2) Limit output to text files of reasonable line lengths and printable characters and include no device-specific formatting information, such as a page description language. The meaning of ‘‘reasonable’’ in this context can only be answered as a quality of implementation issue, but should be apparent from historical usage patterns in the industry and the locale. The `pr` and `fold` utilities can be used to achieve reasonable formatting for the implementation's default page size.

Alternatively, the application can arrange its installation in such a way that requires the system administrator or operator to provide the appropriate information on `lp` options and environment variable values.

At a minimum, having this utility in the standard tells the industry that portable applications require a means to print output and provides at least a command name and **LPDEST** routing mechanism that can be used for discussions between vendors, application writers, and users. The use of ‘‘should’’ in the Description clearly shows the working group's intent, even if it cannot mandate that all systems (such as laptops) have printers.

Examples:

To print file *file*:

```
lp −c file
```

To print multiple files with headers:

```
pr file1 file2 | lp
```

On most existing implementations of `lp`, an option is provided to pass printer specific options to the daemon handling the printer. It is not specified here because the printer-specific options are widespread and in conflict, the `lp` specified here is not required to even have a queueing mechanism, and the choice of options varies widely from printer to printer. Nonetheless, implementors are encouraged to use this mechanism where appropriate:

`−o` *option* Specifies an implementation-defined option that controls the specific operation of the printer. The following *option* s could be used for the meanings below if the hardware is capable of supporting the option.

option	**Meaning**
`lp2`	two logical pages per physical page
`lp4`	four logical pages per physical page
`d`	double sided

POSIX. 2 does not specify what the ownership of the process performing the writ- 1
ing to the output device may be. If `−c` is not used, it is unspecified whether the 1
process performing the writing to the output device will have permission to read 1
file if there are any restrictions in place on who may read *file* until after it is 1
printed. Also, if `−c` is not used, the results of deleting *file* before it is printed are 1
unspecified. 1

History of Decisions Made

The `lp` utility was designed to be a basic version of a utility that is already available in many historical implementations. The working group felt that it should be implementable simply as:

```
cat "$@" > /dev/lp
```

after appropriate processing of options, if that is how the implementation chose to do it and if exclusive access could be granted (so that two users did not write to the device simultaneously). Although in the future the working group may add other options to this utility, it should always be able to execute with no options or operands and send the standard input to an unspecified output device.

The standard makes no representations concerning the format of the printed output, except that it must be ‘‘human-readable’’ and ‘‘nonvolatile.’’ Thus, writing by default to a disk or tape drive or a display terminal would not qualify. (Such destinations are not prohibited when `−d` *dest* , **LPDEST** , or **PRINTER** are used, however.)

A portable application will 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.

The standard is worded such that a ‘‘print job’’ consisting of multiple input files, possibly in multiple copies, is guaranteed to print so that any one file is not jumbled up with another, but there is no statement that all the files or copies have to print out together.

The `-c` option may imply a spooling operation, but this is not required. The utility can be implemented to simply wait until the printer is ready and then wait until it's finished. Because of that, there is no attempt to define a queueing mechanism (priorities, classes of output, etc.).

The `-n` and `-d` options were added in response to balloting objections that too little historical value was being provided.

Although the historical System V `lp` and BSD `lpr` utilities have provided similar functionality, they used different names for the environment variable specifying the destination printer. Since the name of the utility here is `lp`, **LPDEST** (used by the System V `lp` utility) was given precedence over **PRINTER** (used by the BSD `lpr` utility). Since environments of users frequently contain one or the other environment variable, the `lp` utility is required to recognize both. If this was not done, many applications would send output to unexpected output devices when users moved from system to system.

Some have commented that `lp` has far too little functionality to make it worthwhile. Requests have proposed additional options or operands or both that added functionality. The requests included:

— wording *requiring* the output to be ''hardcopy''

— a requirement for multiple printers

— options for PostScript, dimpress, hp, and lineprint formats

Given that a POSIX.2 compliant system is not required to even have a printer, placing further restrictions upon the behavior of the printer is not useful. Since hardcopy format is so application dependent, it is difficult, if not impossible, to select a reasonable subset of functionality that should be required on all POSIX.2 compliant systems.

The term ''unspecified'' is used in this clause in lieu of ''implementation defined'' as most known implementations would not be able to say anything fully useful in their conformance documents: the existence and usage of printers is very dependent on how the system administrator configures each individual system.

4.39 `ls` — List directory contents

4.39.1 Synopsis

`ls [−CFRacdilqrtu1][`*file* ...`]`

4.39.2 Description

For each operand that names a file of a type other than directory, `ls` shall write the name of the file as well as any requested, associated information. For each operand that names a file of type directory, `ls` shall write 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 shall be written. If more than one operand is specified, nondirectory operands shall be written first; directory and nondirectory operands shall be sorted separately according to the collating sequence in the current locale.

4.39.3 Options

The `ls` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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 (`*`) after each that is executable, and a vertical bar (`|`) after each that is a FIFO.

`−R` Recursively list subdirectories encountered.

`−a` Write out all directory entries, including those whose names begin with a period (`.`). Entries beginning with a period (.) shall not be written out unless explicitly referenced, the `−a` option is supplied, or an implementation-defined condition causes them to be written.

`−c` Use time of last modification of the file status information (see POSIX.1 {8} 5.6.1.3) instead of last modification of the file itself for sorting (`−t`) or writing (`−l`).

`−d` Do not treat directories differently than other types of files. The use of `−d` with `−R` produces unspecified results. 2

`−i` For each file, write the file's file serial number (see POSIX.1 {8} 5.6.2).

`−l` (The letter ell.) Write out in long format (see 4.39.6.1). When `−l` 2
(ell) is specified, `−1` (one) shall be assumed. 2

`−q` Force each instance of nonprintable filename characters and 2
`<tab>`s 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.

`−t` Sort by time modified (most recently modified first) before sorting the operands by the collating sequence.

`−u` Use time of last access (see POSIX.1 {8} 5.6.1.3) instead of last modification of the file for sorting (`−t`) or writing (`−l`).

`−1` (The numeric digit one.) Force output to be one entry per line.

Specifying more than one of the options in the following mutually exclusive pairs 2
shall not be considered an error: `−C` and `−l` (ell), `−C` and `−1` (one), `−c` and `−u`. The 2
last option specified in each pair shall determine the output format. 2

4.39.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a file to be written. If the file specified is not found, a diagnostic message shall be output on standard error.

4.39.5 External Influences

4.39.5.1 Standard Input

None.

4.39.5.2 Input Files

None.

4.39.5.3 Environment Variables

The following environment variables shall affect the execution of `ls`:

COLUMNS This variable shall 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 shall calculate how many pathname text columns to write (see −C) based on the width provided. If **COLUMNS** is not set or invalid, an implementation-defined number of column positions shall be 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 shall be constant. File names shall not be truncated to fit into the multiple-text-column output.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for character collation information in determining the pathname collation sequence.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and which characters are defined as printable (character class `print`).

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

LC_TIME This variable shall determine the the format and contents for date and time strings written by `ls`.

TZ This variable shall determine the time zone for date and time strings written by `ls`.

4.39.5.4 Asynchronous Events

Default.

4.39.6 External Effects

4.39.6.1 Standard Output

The default format shall be to list one entry per line to standard output; the exceptions are to terminals or when the −C option is specified. If the output is to a terminal, the format is implementation defined.

2

If the `-i` option is specified, the file's file serial number (see POSIX.1 {8} 5.6.1) shall be written in the following format before any other output for the 2
corresponding entry: 2

`"%u "`, *<file serial number>* 2

If the `-l` option is specified, the following information shall be written:

`"%s %u %s %s %u %s %s\n"`, *<file mode>*, *<number of links>*, 1
<owner name>, *<group name>*, *<number of bytes in the file>*,
<date and time>, *<pathname>*

If *<owner name>* or *<group name>* cannot be determined, they shall be replaced with their associated numeric values using the format `"%u"`.

The *<date and time>*, field shall contain the appropriate date and time stamp of when the file was last modified. In the POSIX Locale, the field shall be the equivalent of the output of the following `date` command (see 4.15):

```
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>` produced by `date` shall not be included and the output shall be 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-defined information associated with the device in question.

If the pathname was specified as a *file* operand, it shall be written as specified.

The file mode written under the `-l` option shall consist 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>* shall be a single `<space>` if there is no alternate or additional access control method associated with the file; otherwise, a printable character shall be used.

The <*entry type*> character shall describe 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-defined, file types.

The next three fields shall be three characters each:

<*owner permissions*> Permissions for the file owner class (see 2.9.1.3).

<*group permissions*> Permissions for the file group class.

<*other permissions*> Permissions for the file other class.

Each field shall have 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 shall, however, be written in lowercase if the file is executable or searchable, and in uppercase if it is not.

If the `–l` option is specified, each list of files within the directory shall 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 shall be:

`"total %u\n"`, <*number of units in the directory*>

If more than one directory, or a combination of nondirectory files and directories are written, either as a result of specifying multiple operands, or the `–R` option, each list of files within a directory shall be preceded by:

`"\n%s:\n"`, <*directory name*>

If this string is the first thing to be written, the first `<newline>` character shall not be written. This output shall precede the number of units in the directory.

4.39.6.2 Standard Error

Used only for diagnostic messages.

4.39.6.3 Output Files

None.

4.39.7 Extended Description

None.

4.39.8 Exit Status

The `ls` utility shall exit with one of the following values:

- 0 All files were written successfully.
- >0 An error occurred.

4.39.9 Consequences of Errors

Default.

4.39.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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
```

Many implementations use the equals-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. These characters should not be used to signify any other types of files in new implementations.

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 (see 4.62).

The output of `ls` (with the `−l` option) 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. In POSIX.2, a character was 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. POSIX.6 may define one or more specific characters to be used based on different standard additional or alternative access control mechanisms.

Some historical implementations of the `ls` utility show all entries in a directory except dot and dot-dot when super-user invokes `ls` without specifying the `−a` option. When ‘‘normal’’ users invoke `ls` without specifying `−a`, they should not see information about any files with names beginning with period unless they were named as file operands.

As with many of the utilities that deal with file names, the output of `ls` for multiple files or in one of the long listing formats must be used carefully on systems where file names can contain embedded white space. It is recommended that systems and system administrators institute policies and user training to limit the use of such file names.

History of Decisions Made

Implementations are expected to traverse arbitrary depths when processing the `−R` option. The only limitation on depth should be based on running out of physical storage for keeping track of untraversed directories.

The `−1` (one) option is currently found in BSD and BSD-derived implementations only. It was required in the standard so that portable applications might ensure that output is one entry per line, even if the output is to a terminal. Recent changes to the 2.10.2 allow numeric options.

Generally, the standard is mute about what happens when options are given multiple times. In the case of `−C`, `−l`, and `−1`, however, it does specify the results of these overlapping options. Since `ls` is one of the most aliased commands, it is important that the implementation do the correct thing. For example, if the alias were

```
alias ls="ls -C"
```

and the user typed ‘`ls -1`’, single text column output should result, not an error.

(The working group is aware that aliases are not included in the standard; this is just an example.)

The *SVID* defines a `-x` option for multi-text-column output sorted horizontally. The working group felt that `-x` provided only limited increased functionality over the `-C` option. The *SVID* also provides a `-m` option for a comma separated list of files. It was not provided because similar functionality (easier to parse for scripts) can be provided by the `echo` and `printf` utilities. Nonetheless, implementations considering adding new options to `ls` should look at historical BSD and System V versions of `ls` to avoid naming conflicts.

The BSD `ls` provides a `-A` option (like `-a`, but dot and dot-dot are not written out). The small difference from `-a` did not seem important enough to require both.

Implementations are allowed to make `-q` the default for terminals to prevent Trojan Horse attacks on terminals with special escape sequences. This is not required because:

— Some control characters may be useful on some terminals; for example, a system might write them as `\001` or `^A`,

— Special behavior for terminals is not relevant to application portability.

The `-s` option provided by existing implementations is not required by this standard. 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 POSIX.1 {8} and not readily accessible on some file system types. Therefore, applications cannot depend on `-s` to provide any portable information. Implementations are urged to continue to provide this option, but applications should use the file size reported by the `-l` option in any calculations about the space needed to store a file.

An earlier draft specified that the optional alternate access method flag had to be ‘‘+’’if there was an alternate access method used on the file or `<space>` if there was not. This was changed in Draft 10 to be `<space>` if there is not and a single printable character if there is. This was done for three reasons: 1) There are existing implementations using characters other than ‘‘+’’; 2) There are implementations that vary this character used in that position to distinguish between various alternate access methods in use, and; 3) the developers of the standard did not want to preclude specification by POSIX.6 that might need a way to specify more than one alternate access method. Nonetheless, implementations providing a single alternate access method are encouraged to use ‘‘+’’.

In a previous draft the units used to specify the number of blocks occupied by files in a directory in an `ls -l` listing was implementation defined. This was because BSD systems have historically used 1024-byte units and System V systems have historically used 512-byte units. It was pointed out by developers at Berkeley that BSD has used 512-byte units in some places and 1024-byte units in other places. (System V has consistently used 512.) Therefore, POSIX.2 and POSIX.2a usually specify 512 and that value has been restored here as it was in Draft 9. Future releases of BSD are expected to consistently provide 512 as a default with a way of specifying 1024-byte units where appropriate.

The <*date and time*> field in the `-l` format is specified only for the POSIX Locale. As noted, the format can be different in other locales. No mechanism for defining this is present in this standard, as the appropriate vehicle is a messaging system; i.e., the format should be specified as a ''message.''

4.40 `mailx` — Process messages

4.40.1 Synopsis

`mailx [-s` *subject* `]` *address* ...

4.40.2 Description

The `mailx` utility shall read standard input and send it to one or more addresses in an unspecified manner. Unless the first character of one or more lines is tilde (~), all characters in the input message shall appear in the delivered message, but additional characters may be inserted in the message before it is retrieved.

4.40.3 Options

The `mailx` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`-s` *subject* A string representing the subject of the message. All characters 2
in the *subject* string shall appear in the delivered message. The 2
results are unspecified if *subject* is longer than {LINE_MAX} − 10 2
bytes or contains a `<newline>`. 2

4.40.4 Operands

The following operand shall be supported by the implementation:

address Send a message to *address*. Valid login names on the local system shall be accepted as valid *address* es. The interpretation of other types of *address* es is unspecified. An implementation-defined way for a user with a login-name address to retrieve the message shall be provided by the implementation.

4.40.5 External Influences

4.40.5.1 Standard Input

The standard input shall be a text file. The results are unspecified if the first character of any input line is a tilde (`~`).

4.40.5.2 Input Files

None.

4.40.5.3 Environment Variables

The following environment variables shall affect the execution of `mailx`:

DEAD This variable shall affect the processing of signals by `mailx`: if the application sets this variable to `/dev/null`, the results of receiving a signal are as described by this standard; they are otherwise unspecified.

HOME This variable shall be interpreted as a pathname of the user's home directory.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

MAILRC This variable shall affect the startup processing of `mailx`: if the application sets this variable to `/dev/null`, `mailx` shall operate as described by this standard; otherwise, unspecified results occur.

4.40.5.4 Asynchronous Events

Default.

4.40.6 External Effects

4.40.6.1 Standard Output

None.

4.40.6.2 Standard Error

Used only for diagnostic messages.

4.40.6.3 Output Files

None.

4.40.7 Extended Description

None.

4.40.8 Exit Status

The `mailx` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

4.40.9 Consequences of Errors

Default.

4.40.10 Rationale. *(This subclause is not a part of P1003.2)*

Usage, Examples

The intent is that a header indicating who sent the message and a message subject string, the contents of the standard input, and perhaps a trailer is delivered to users specified by the given addresses. The standard input, however, may have to be manipulated slightly to avoid confusion between message text and headers as it passes through the message delivery system. POSIX.2 does not specify how standard input may be manipulated; that will be specified in detail by POSIX.2a.

The restriction on a subject line being {LINE_MAX} − 10 bytes is based on the his- 2
torical format that consumes 10 bytes for `"Subject: "` and the trailing <new- 2
line>. Many historical mailers that a message may encounter on other systems 2
will not be able to handle lines that long, however. 2

History of Decisions Made

The developers of the standard felt strongly that a method for applications to send messages to specific users was necessary. The obvious example is a batch utility, running noninteractively, that wishes to communicate errors or results to a user. However, the actual format, delivery mechanism, and method of reading the message are clearly beyond the scope of this standard.

The intent of this command is to provide a simple, portable interface for sending messages noninteractively. It merely defines a ‘‘front-end’’ to the historical mail system. It is suggested that implementations explicitly denote the sender and recipient in the body of the delivered message. Further specification of formats for either the message envelope or the message itself were deliberately not made, as the industry is in the midst of changing from the current standards to a more internationalized standard and it is probably incorrect, at this time, to require either one.

Implementations are encouraged to conform to the various delivery mechanisms described in ARPANET Requests for Comment Numbers 819, 822, 882, 920, 921, and the CCITT X.400 standards.

The standard 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. (Note that this is usually an administrative issue based on the amount of mail traffic and disk space available on the gateways. Therefore, there is no way for this standard to require implementations to guarantee delivery of long messages to remote systems.)

Like the utilities `logger` and `lp`, `mailx` is admittedly difficult to test. This was not deemed sufficient justification to exclude these utilities from the standard. It is also arguable that they are, in fact, testable, but that the tests themselves are not portable.

Before Draft 7, there was a utility named `mailto`. In Draft 7, the name was changed to `sendto` because of comments noting that `mailto` implied full mail-like functionality and that was not what the specification provided. However, there have been consistent comments that it does not make sense to end up with a standard that will require two mail-sending interfaces. (POSIX.2a is working on a fully fleshed-out mail-sending and -reading utility based on the historical System V `mailx` utility.) A message- (or mail-) sending utility that is a subset of the interactive utility that will be described by POSIX.2a is much more consistent with the rest of the standard. Therefore, in Draft 10 the name has been changed again to `mailx` and the description is a small subset of the functionality being specified by POSIX.2a. It provides a portable way for a shell script to be able to send a message to a user on the local system. It is expected that implementations that have provided `mailx` in the past will use it to meet the POSIX.2 requirements. Implementations that have not provided `mailx` in the past will be able to create a simple interface to their current mailer to meet these requirements.

Most of the features provided by `mailx` (and the similar BSD `Mail)` utility are not specified here because they are not needed for noninteractive use (applications do not usually read mail without user participation) and they depend on other interactive features that are not defined by POSIX.2, but will be defined by POSIX.2a (the `~v` command, for instance, uses the `vi` editor as a default.)

If the **DEAD** environment variable is not set to `/dev/null`, historical versions of `mailx` and `Mail` save a message being constructed in a file under some circumstances when some asynchronous events occur. The details will be specified by POSIX.2a.

If the **MAILRC** environment variable does not name an empty file, historical versions of `mailx` and `Mail` read initialization commands from a file before processing begins. Since the initialization that a user specifies could alter the contents of messages an application is trying to send, applications are advised to set **MAILRC** to `/dev/null`. POSIX.2a will specify details on the format of the initialization file.

Options to specify addresses as ‘‘cc’’(carbon-copy) or ‘‘bcc’’(blind-carbon-copy) were considered to be format details and were omitted.

A zero exit status implies that all messages were *sent*, but it gives no assurances that any of them were actually *delivered*. The reliability of the delivery mechanism is unspecified and is an appropriate marketing distinction between systems.

4.41 `mkdir` — Make directories

4.41.1 Synopsis

`mkdir [−p][−m` *mode* `]` *dir* . . .

4.41.2 Description

The `mkdir` utility shall create the directories specified by the operands, in the order specified.

For each *dir* operand, the `mkdir` utility shall perform actions equivalent to the POSIX.1 {8} *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, 1
the *mode* option-argument overrides this default.) 1

4.41.3 Options

The `mkdir` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−m` *mode* Set the file permission bits of the newly-created directory to the specified *mode* value. The *mode* option-argument shall be the same as the *mode* operand defined for the `chmod` utility (see 4.7). In the *symbolic_mode* strings, the *op* characters `+` and `-` shall be interpreted relative to an assumed initial mode of `a=rwx`; `+` shall add permissions to the default mode, `-` shall 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 following command shall 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 shall be ignored without error.

4.41.4 Operands

The following operand shall be supported by the implementation:

dir A pathname of a directory to be created.

4.41.5 External Influences

4.41.5.1 Standard Input

None.

4.41.5.2 Input Files

None.

4.41.5.3 Environment Variables

The following environment variables shall affect the execution of `mkdir`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.41.5.4 Asynchronous Events

Default.

4.41.6 External Effects

4.41.6.1 Standard Output

None.

4.41.6.2 Standard Error

Used only for diagnostic messages.

4.41.6.3 Output Files

None.

4.41.7 Extended Description

None.

4.41.8 Exit Status

The `mkdir` utility shall exit with one of the following values:

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.

4.41.9 Consequences of Errors

Default.

4.41.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The default file mode for directories is `a=rwx` (777) with selected permissions removed in accordance with the file mode creation mask. For intermediate path name 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 afterward with `chmod`.

Application writers should note that some of the requested directories may have been created even if an error occurs.

History of Decisions Made

The System V `-m` option was added to control the file mode.

The System V `-p` option was added to create any needed intermediate directories, to complement the functionality provided `rmdir` for removing directories in the path prefix as they become empty. Because no error is produced if any path component already exists, the `-p` option is also useful to ensure that a particular directory exists.

The functionality of `mkdir` is described substantially through a reference to the *mkdir* () function in POSIX.1 {8}. For example, by default, the mode of the directory is affected by the file mode creation mask in accordance with the specified behavior of POSIX.1 {8} *mkdir* (). In this way, there is less duplication of effort required for describing details of the directory creation.

4.42 mkfifo — Make FIFO special files

4.42.1 Synopsis

```
mkfifo [−m mode ] file ...
```

4.42.2 Description

The mkfifo utility shall create the FIFO special files specified by the operands, in the order specified.

For each *file* operand, the mkfifo utility shall perform actions equivalent to the POSIX.1 {8} *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.)

4.42.3 Options

The mkfifo utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

−m *mode* Set the file permission bits of the newly-created FIFO to the specified *mode* value. The *mode* option-argument shall be the same as the *mode* operand defined for the chmod utility (see 4.7). In the *symbolic_mode* strings, the *op* characters + and - shall be interpreted relative to an assumed initial mode of a=rw.

4.42.4 Operands

The following operand shall be supported by the implementation:

file A pathname of the FIFO special file to be created.

4.42.5 External Influences

4.42.5.1 Standard Input

None.

4.42.5.2 Input Files

None.

4.42.5.3 Environment Variables

The following environment variables shall affect the execution of `mkfifo`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.42.5.4 Asynchronous Events

Default.

4.42.6 External Effects

4.42.6.1 Standard Output

None.

4.42.6.2 Standard Error

Used only for diagnostic messages.

4.42.6.3 Output Files

None.

4.42.7 Extended Description

None.

4.42.8 Exit Status

The `mkfifo` utility shall exit with one of the following values:

0 All the specified FIFO special files were created successfully.

>0 An error occurred.

4.42.9 Consequences of Errors

Default.

4.42.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

This new utility was added to permit shell applications to create FIFO special files.

The `-m` option was added to control the file mode, for consistency with the similar functionality provided the `mkdir` utility.

Earlier drafts included a `-p` option similar to `mkdir`'s `-p` option that created intermediate directories leading up to the FIFO specified by the final component. This was removed because it is not commonly needed and is not common practice with similar utilities.

The functionality of `mkfifo` is described substantially through a reference to the *mkfifo* () function in POSIX. 1. For example, by default, the mode of the FIFO file is affected by the file mode creation mask in accordance with the specified behavior of POSIX .1 {8} *mkfifo* (). In this way, there is less duplication of effort required for describing details of the file creation.

4.43 `mv` — Move files

4.43.1 Synopsis

`mv [-fi]` *source_file target_file*

`mv [-fi]` *source_file* ... *target_dir*

4.43.2 Description

In the first synopsis form, the `mv` utility shall move 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` shall move 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* shall be the concatenation of the target directory, a single slash character, and the last pathname component of the *source_file*.

If any operand specifies an existing file of a type not specified by POSIX.1 {8}, the behavior is implementation defined.

This second form is assumed when the final operand names an existing directory.

For each *source_file* the following steps shall be 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 shall write a prompt to standard error and read a line from standard input. If the response is not affirmative, `mv` shall do nothing more with the current *source_file* and go on to any remaining *source_file* s.

(2) The `mv` utility shall perform actions equivalent to the POSIX.1 {8} *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` shall 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 POSIX.1 {8}, `mv` shall 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 shall 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 shall attempt to remove it. If this fails for any reason, mv shall 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* shall be duplicated as a file hierarchy rooted in the destination path. The following characteristics of each file in the file hierarchy shall be duplicated:

(a) The time of last data modification and time of last access.

(b) The user ID and group ID.

(c) 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 shall not be duplicated.

When files are duplicated to another file system, the implementation may 1
require that the process invoking mv have read access to each file being 1
duplicated. 1

If the duplication of the file hierarchy fails for any reason, mv shall 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 shall write a diagnostic message to standard error, but this failure shall not cause mv to modify its exit status.

(6) The file hierarchy rooted in *source_file* shall be removed. If this fails for any reason, mv shall 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.

4.43.3 Options

The mv utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−f Do not prompt for confirmation if the *destination* path exists. Any previous occurrences of the −i option shall be ignored.

`-i` Prompt for confirmation if the destination path exists. Any previous occurrences of the `-f` option shall be ignored.

Specifying more than one of the `-f` or `-i` options shall not be considered an error. The last option specified shall determine `mv`'s behavior.

4.43.4 Operands

The following operands shall be supported by the implementation:

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.

4.43.5 External Influences

4.43.5.1 Standard Input

Used to read an input line in response to each prompt specified in Standard Error. 4.43.6.2. Otherwise, the standard input shall not be used.

4.43.5.2 Input Files

The input files specified by each *source_file* operand can be of any file type.

4.43.5.3 Environment Variables

The following environment variables shall affect the execution of `mv`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior 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 This variable shall determine the processing of affirmative responses and the language in which messages should be written.

4.43.5.4 Asynchronous Events

Default.

4.43.6 External Effects

4.43.6.1 Standard Output

None.

4.43.6.2 Standard Error

Prompts shall be written to the standard error under the conditions specified in 4.43.2. The prompts shall contain the *destination* pathname, but their format is otherwise unspecified. Otherwise, the standard error shall be used only for diagnostic messages.

4.43.6.3 Output Files

The output files may be of any file type.

4.43.7 Extended Description

None.

4.43.8 Exit Status

The `mv` utility shall exit with one of the following values:

0 All input files were moved successfully.

>0 An error occurred.

4.43.9 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 shall not modify both *source_file* and the destination path simultaneously; termination at any point shall leave either *source_file* or the destination path complete.

4.43.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

If the current directory contains only files `a` (of any type defined by POSIX.1 {8}), `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.

History of Decisions Made

Previous versions of this draft diverged from *SVID* and BSD historical practice in that they required that when the destination path exists, the `−f` option is not specified, and input is not a terminal, `mv` shall fail. This was done for compatibility with `cp`. This draft returns to historical practice. It should be noted that this is consistent with the POSIX.1 {8} function *rename*(), which does not require write permission on the target.

For absolute clarity, paragraph (1), describing `mv`'s behavior when prompting for confirmation, should be interpreted in the following manner:

```
if (exists AND (NOT f_option) AND
        ((not_writable AND input_is_terminal) OR i_option))
```

The `−i` option exists on BSD systems, giving applications and users a way to avoid accidentally unlinking files when moving others. When the standard input is not a terminal, the 4.3BSD `mv` deletes all existing destination paths without prompting, even when `−i` is specified; this is inconsistent with the behavior of the 4.3BSD `cp` utility, which always generates an error when the file is unwritable and the standard input is not a terminal. The working group decided that use of `−i` is a request for interaction, so when the *destination* path exists, the utility takes instructions from whatever responds to standard input.

The *rename*() function is able to move directories within the same file system. 1
Some historical versions of `mv` have been able to move directories, but not to a dif- 1
ferent file system. The working group felt that this was an annoying inconsistency, so the standard requires directories to be movable even across file systems. There is no `−R` option to confirm that moving a directory is actually intended, since such an option was not required for moving directories in historical practice. Requiring the application to specify it sometimes, depending on the

destination, seemed just as inconsistent. The semantics of the *rename* () function were preserved as much as possible. For example, `mv` is not permitted to ‘‘rename’’ files to or from directories, even though they might be empty and removable.

Historic implementations of `mv` did not exit with a nonzero exit status if they were unable to duplicate any file characteristics when moving a file across file systems, nor did they write a diagnostic message for the user. The former behavior has been preserved to prevent scripts from breaking; a diagnostic message is now required, however, so that users are alerted that the file characteristics have changed.

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified, because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application not using the `−f` option or using the `−i` option relies on the system to provide the most suitable dialogue directly with the user, based on the behavior specified.

4.44 nohup — Invoke a utility immune to hangups

4.44.1 Synopsis

`nohup` *utility* [*argument* . . .]

4.44.2 Description

The `nohup` utility shall 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 shall be set to be ignored.

If the standard output is a terminal, all output written by the named *utility* to its standard output shall 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 shall 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* shall not be invoked. If a file is created, the file's permission bits shall be set to S_IRUSR | S_IWUSR instead of the default specified in 2.9.1.4.

If the standard error is a terminal, all output written by the named *utility* to its standard error shall be redirected to the same file descriptor as the standard output.

4.44.3 Options

None.

4.44.4 Operands

The following operands shall be supported by the implementation:

utility The name of a utility that is to be invoked. If the *utility* operand names any of the special built-in utilities in 3.14, the results are undefined.

argument Any string to be supplied as an argument when invoking the utility named by the *utility* operand.

4.44.5 External Influences

4.44.5.1 Standard Input

None.

4.44.5.2 Input Files

None.

4.44.5.3 Environment Variables

The following environment variables shall affect the execution of `nohup`:

HOME This variable shall 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 shall use the directory named by **HOME** to create the file.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

PATH This variable shall determine the search path that shall be used to locate the utility to be invoked. See 2.6.

4.44.5.4 Asynchronous Events

The `nohup` utility shall take the standard action for all signals (see 2.11.5.4), except that SIGHUP shall be ignored.

4.44.6 External Effects

4.44.6.1 Standard Output

If the standard output is not a terminal, the standard output of `nohup` shall be the standard output generated by the execution of the *utility* specified by the operands. Otherwise, nothing shall be written to the standard output.

4.44.6.2 Standard Error

If the standard output is a terminal, a message shall be written to the standard error, indicating the name of the file to which the output is being appended. The name of the file shall be either `nohup.out` or `$HOME/nohup.out`.

4.44.6.3 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.

4.44.7 Extended Description

None.

4.44.8 Exit Status

The `nohup` utility shall exit with one of the following values:

126 The utility specified by *utility* was found but could not be invoked. 1

127 An error occurred in the `nohup` utility or the utility specified by *utility* 1
could not be found. 1

Otherwise, the exit status of `nohup` shall be that of the utility specified by the *utility* operand.

4.44.9 Consequences of Errors

Default.

4.44.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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*`'`

The 4.3BSD version ignores SIGTERM and SIGHUP, and if `./nohup.out` cannot be used, it fails instead of trying to use `$HOME/nohup.out`.

The `command`, `env`, `nohup`, and `xargs` utilities have been specified to use exit
code 127 if an error occurs so that applications can distinguish ‘‘failure to find a 1
utility’’ from ‘‘invoked utility exited with an error indication.’’ The value 127 was 1
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 con-
fused with termination due to receipt of a signal. The value 126 was chosen in a 1
similar manner to indicate that the utility could be found, but not invoked. Some 1
scripts produce meaningful error messages differentiating the 126 and 127 cases. 1
The distinction between exit codes 126 and 127 is based on KornShell practice 2
that uses 127 when all attempts to *exec* the utility fail with [ENOENT], and uses 2
126 when any attempt to *exec* the utility fails for any other reason. 2

History of Decisions Made

The `csh` utility has a built-in version of `nohup` that acts differently than this.

The term *utility* is used, rather than *command*, to highlight the fact that shell compound commands, pipelines, special built-ins, etc., cannot be used directly. However, *utility* includes user application programs and shell scripts, not just the standard utilities.

Historical versions of the `nohup` utility use default file creation semantics. Some more recent versions use the permissions specified here as an added security precaution.

Some historical implementations ignore SIGQUIT in addition to SIGHUP; others ignore SIGTERM. An earlier draft allowed, but did not require, SIGQUIT to be ignored. Several members of the balloting group objected, saying that `nohup` should only modify the handling of SIGHUP as required by this specification.

4.45 `od` — Dump files in various formats

4.45.1 Synopsis

`od` [`−v`] [`−A` *address_base*] [`−j` *skip*] [`−N` *count*] [`−t` *type_string*] . . . [*file* . . .]

4.45.2 Description

The `od` utility shall write the contents of its input files to standard output in a user-specified format.

4.45.3 Options

The `od` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that the order of presentation of the `−t` options is significant.

The following options shall be supported by the implementation:

`−A` *address_base*
: Specify the input offset base (see 4.45.7). The *address_base* option argument shall be a character. The characters `d`, `o`, and `x` shall specify that the offset base shall be written in decimal, octal, or hexadecimal, respectively. The character `n` shall specify that the offset shall not be written.

`−j` *skip*
: Jump over *skip* bytes from the beginning of the input. The `od` utility shall 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 shall write a diagnostic message to standard error and exit with a nonzero exit status.

: By default, the *skip* option-argument shall be interpreted as a decimal number. With a leading `0x` or `0X`, the offset shall be interpreted as a hexadecimal number; otherwise, with a leading `0`, the offset shall be interpreted as an octal number. Appending the character `b`, `k`, or `m` to offset shall cause it to be interpreted as a multiple of 512, 1024, or 1 048 576 bytes, respectively.

`−N` *count*
: Format no more than *count* bytes of input. By default, *count* shall be interpreted as a decimal number. With a leading `0x` or `0X`, *count* shall be interpreted as a hexadecimal number; otherwise, with a leading `0`, it shall be interpreted as an octal number. If *count* bytes of input (after successfully skipping, if `−j` *skip* is specified) are not available, it shall not be considered an error; the `od` utility shall format the input that is available.

`−t` *type_string*

Specify one or more output types (see 4.45.7). The *type_string* option-argument shall be a string specifying the types to be used when writing the input data. The string shall 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 shall be 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), shall be replaced with a line containing only an asterisk (*).

4.45.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a file to be written. If no file operands are specified, the standard input shall be used. The results are unspecified if the first character of *file* is a plus-sign (+) or the first character of the first file operand is numeric, unless at least one of the `−A`, `−j`, `−N`, or `−t` options is specified.

4.45.5 External Influences

4.45.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.45.5.2 Input Files

The input files can be any file type.

4.45.5.3 Environment Variables

The following environment variables shall affect the execution of `od`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

LC_NUMERIC This variable shall determine the locale for selecting the radix character used when writing floating-point formatted output.

4.45.5.4 Asynchronous Events

Default.

4.45.6 External Effects

4.45.6.1 Standard Output

See 4.45.7.

4.45.6.2 Standard Error

Used only for diagnostic messages. 2

4.45.6.3 Output Files

None.

4.45.7 Extended Description

The `od` utility shall copy sequentially each input file to standard output, transforming the input data according to the output types specified by the `-t` option(s). If no output type is specified, the default output shall be 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` shall correspond to the various C-language types as follows. If the `c89` compiler is present on the system, these specifiers shall correspond to the sizes used by default in that compiler. Otherwise, these sizes are implementation defined.

— For the type specifier characters `d`, `o`, `u`, and `x`, the default number of bytes shall correspond to the size of the underlying implementation's basic integral data type. For these specifier characters, the implementation shall 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 defined, but shall 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 shall correspond to the number of bytes in the underlying implementation's basic double precision floating point data type. The implementation shall 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 shall be interpreted as named characters from the International Reference Version (IRV) of ISO/IEC 646 {1}. Only the least significant seven bits of each byte shall be used for this type specification. Bytes with the values listed in Table 4-8 shall be written using the corresponding names for those characters.

The type specifier character `c` specifies that bytes shall be interpreted as characters specified by the current setting of the LC_CTYPE locale category. Characters listed in Table 2-15 (see 2.12) shall be written as the corresponding escape sequences, except that backslash shall be written as a single backslash and a NUL shall be written as `\0`. Other nonprintable characters shall 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 nonprintable characters is implementation-defined. Printable multibyte characters shall be written in the area corresponding to the first byte of the character; the two-character sequence ** shall be written in the area corresponding to each remaining byte in the character, as an indication that the character is continued.

Table 4-8 – od Named 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`				

NOTE: The `\012` value may be written either as `lf` or `nl`.

The input data shall be 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 shall 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 shall sequentially transform the parts of the input block and the output from each of the transformations shall be separated by one or more <blank>s.

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 shall 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 shall 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 shall not contain any <blank>s, shall start at the first character of the output line, and shall be followed by one or more <blank>s. In addition, the offset of the byte following the last byte written shall be written after all the input data has been processed, but shall not be followed by any <blank>s.

If no −A option is specified, the input offset base is unspecified.

4.45.8 Exit Status

The `od` utility shall exit with one of the following values:

- 0 All input files were processed successfully.
- >0 An error occurred.

4.45.9 Consequences of Errors

Default.

4.45.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

If a file containing 128 bytes with decimal values zero through 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 standard allows `nl` or `lf` to be used as the name for the ISO/IEC 646 {1} IRV character with decimal value 10. The IRV names this character `lf` (line feed), but traditional implementations on which POSIX.2 are based have referred to this character as newline (`nl`) and the POSIX Locale character set symbolic name for the corresponding character is `<newline>`.

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    402f7851    eb851eb8
0000016    1.40668230000000e+02
        10030312542 04370303230
           40619562    23e18698
0000024
```

History of Decisions Made

The `od` utility has gone through several names in previous drafts, including `hd`, `xd`, and most recently `hexdump`. There were several objections to all of these based on the following reasons:

— The `hd` and `xd` names conflicted with existing utilities that behaved differently.

— The `hexdump` description was much more complex than needed for a simple dump utility.

— The `od` utility has been available on all traditional implementations and there was no need to create a new name for a utility so similar to the existing `od` utility.

The original reasons for not standardizing historical `od` were also fairly widespread. Those reasons are given below along with rationale explaining why the developers of this standard believe that this version does not suffer from the indicated problem:

— The BSD and System V versions of `od` have diverged and the intersection of features provided by both does not meet the needs of the user community. In fact, the System V version only provides a mechanism for dumping octal bytes and *short* s, signed and unsigned decimal *short* s, hexadecimal *short* s, and ASCII characters. BSD added the ability to dump *float* s, *double* s, named ASCII characters, and octal, signed decimal, unsigned decimal, and hexadecimal *long* s. The version presented here provides more normalized forms for dumping bytes, *short* s, *int* s, and *long* s in octal, signed decimal, unsigned decimal, and hexadecimal; *float, double,* and *long double* ; and named ASCII as well as current locale characters.

— It would not be possible to come up with a compatible superset of the BSD and System V flags that met the requirements of this standard. The historical default `od` output is the specified default output of this utility. None of the option letters chosen for this version of `od` conflict with any of the options to historical versions of `od`.

— On systems with different sizes for *short, int,* and *long,* there was no way to ask for dumps of *int* s, even in the BSD version. The way options are named, there is no easy way to extend the namespace for these problems. This is why the `-t` option was added with type specifiers more closely matched to the *printf* () formats used in the rest of this standard and the optional field sizes were added to the `d`, `f`, `o`, `u`, and `x` type specifiers. It is also one of the reasons why the historical practice was not mandated as a required obsolescent form of `od`. (Although the old versions of `od` are not listed as an obsolescent form, implementations are urged to continue to recognize the old forms they have recognized for a few years.) The `a`, `c`, `f`, `o`, and `x` types match the meaning of the corresponding format characters in the historical implementations of `od` except for the default sizes of the fields converted. The `d` format is signed in this specification to match the *printf* () notation. (Historical versions of `od` used `d` as a synonym for `u` in this version. The System V implementation uses `s` for signed decimal; BSD uses `i` for signed decimal and `s` for null terminated strings.) Other than `d` and `u`, all of the type specifiers match format characters in the historical BSD version of `od`.

The sizes of the C-language types *char*, *short*, *int*, *long*, *float*, *double*, and *long double* are used even though it is recognized that there may be zero or more than one compiler for the C language on an implementation and that they may use different sizes for some of these types. [For example, one compiler might use 2-byte *shorts,* 2-byte *ints,* and 4-byte *longs* while another compiler (or an option to the same compiler) uses 2-byte *shorts,* 4-byte *ints,* and 4-byte *longs*.] Nonetheless, there has to be a basic size known by the implementation for these types, corresponding to the values reported by invocations of the `getconf` utility (see 4.26) when called with *system_var* operands UCHAR_MAX, USHORT_MAX, UINT_MAX, and ULONG_MAX for the types *char, short, int,* and *long,* respectively. There are similar constants required by the C Standard {7}, but not required by POSIX.1 {8} or POSIX.2. They are FLT_MANT_DIG, DBL_MANT_DIG, and LDBL_MANT_DIG for the types *float, double,* and *long double*, respectively. If the optional `c89` utility (see A.1) is provided by the implementation and used as specified by this standard, these are the sizes that would be provided. If an option is used that specifies different sizes for these types, there is no guarantee that the `od` utility will be able to correctly interpret binary data output by such a program.

POSIX.2 requires that the numeric values of these lengths be recognized by the `od` utility and that symbolic forms also be recognized. Thus a portable application can always look at an array of *unsigned long* data elements using `od -t uL`.

— The method of specifying the format for the address field based on specifying a starting offset in a file unnecessarily tied the two together. The `–A` option now specifies the address base and the `–S` option specifies a starting offset. 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 POSIX.2. To guarantee that one of these filenames will always be interpreted as a file name, an application could always specify the address base format with the `–A` option.

— It would be hard to break the dependence on US ASCII to get an internationalized utility. It does not seem to be any harder for `od` to dump characters in the current locale than it is for the `ed` or `sed l` commands. The `c` type specifier does this with no problem and is completely compatible with the historical implementations of the `c` format character when the current locale uses a superset of ISO/IEC 646 {1} as a code set. The `a` type specifier (from the BSD `a` format character) was left as a portable means to dump ASCII [or more correctly ISO/IEC 646 {1} (IRV)] so that headers produced by `pax` could be deciphered even on systems that do not use ISO/IEC 646 {1} as a subset of their base code set.

The use of ** as an indication of continuation of a multibyte character in `c` specifier output was chosen based on seeing an implementation that uses this method. The continuation bytes have to be marked in a way that will not be ambiguous with another single- or multibyte character.

An earlier draft used `−S` and `−n`, respectively, for the `−j` and `−N` options in this draft. These were changed to avoid conflicts with historical implementations.

4.46 `paste` — Merge corresponding or subsequent lines of files

4.46.1 Synopsis

`paste [−s] [−d` *list* `]` *file* . . .

4.46.2 Description

The `paste` utility shall concatenate the corresponding lines of the given input files, and write the resulting lines to standard output.

The default operation of `paste` shall concatenate the corresponding lines of the input files. The `<newline>` character of every line except the line from the last input file shall 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` shall behave as though empty lines were read from the file(s) on which end-of-file was detected, unless the `−s` option is specified.

4.46.3 Options

The `paste` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−d` *list* Unless a backslash character appears in *list*, each character in 2
list is an element specifying a delimiter character. If a backslash 2
character appears in *list*, the backslash character and one or more 2
characters following it are an element specifying a delimiter char- 2
acter as described below. These elements specify one or more 2

delimiters to use, instead of the default `<tab>`, to replace the 2
`<newline>` character of the input lines. The elements in *list* 2
shall be used circularly; i.e., when the list is exhausted the first 2
element from the list shall be re-used. When the `−s` option is 2
specified:

- The last `<newline>` character in a file shall not be modified.
- The delimiter shall 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 shall not be modified.
- The delimiter shall 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 shall 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 2.5.2.1), 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 shall be replaced with the `<tab>` character, unless otherwise specified by the `−d` option.

4.46.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If `−` is specified for one or more of the *file*s, the standard input shall be used; the standard input shall be read one line at a time, circularly, for each instance of `−`. Implementations shall support pasting of at least 12 *file* operands.

4.46.5 External Influences

4.46.5.1 Standard Input

The standard input shall be used only if one or more *file* operands is –. See Input Files.

4.46.5.2 Input Files

The input files shall be text files, except that line lengths shall be unlimited.

4.46.5.3 Environment Variables

The following environment variables shall affect the execution of `paste`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.46.5.4 Asynchronous Events

Default.

4.46.6 External Effects

4.46.6.1 Standard Output

Concatenated lines of input files shall be separated by the `<tab>` character (or other characters under the control of the `–d` option) and terminated by a `<newline>` character.

4.46.6.2 Standard Error

Used only for diagnostic messages.

4.46.6.3 Output Files

None.

4.46.7 Extended Description

None.

4.46.8 Exit Status

The `paste` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.46.9 Consequences of Errors

If one or more input files cannot be opened when the `−s` option is not specified, a diagnostic message shall be written to standard error, but no output shall be written to standard output. If the `−s` option is specified, the `paste` utility shall provide the default behavior described in 2.11.9.

4.46.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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.

Write out a directory in four columns:

```
ls | paste - - - -
```

Combine pairs of lines from a file into single lines:

```
paste -s -d "\t\n" file
```

Portable applications should only use the specific backslash escaped delimiters presented in this standard. Historical implementations treat `\x`, where `x` is not in this list, as `x`, but future implementations are free to expand this list to recognize 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 POSIX.2 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* ().

History of Decisions Made

Because most of the standards utilities work on text files, `cut` and `paste` are required to process lines of arbitrary length as a means of converting long lines from arbitrary sources into text files and converting processed text files back into files with arbitrary line lengths to interface with those applications that require long lines as input.

4.47 pathchk — Check pathnames

4.47.1 Synopsis

`pathchk [−p]` *pathname* ...

4.47.2 Description

The `pathchk` utility shall check that one or more pathnames are valid (i.e., they could be used to access or create a file without causing syntax errors) and portable (i.e., no filename truncation will result). More extensive portability checks are provided by the `−p` option.

By default, the `pathchk` utility shall check each component of each *pathname* operand based on the underlying file system. A diagnostic shall be written for each *pathname* operand that:

— is longer than {PATH_MAX} bytes (see Pathname Variable Values in POSIX.1 {8}2.9.5),

— contains any component longer than {NAME_MAX} bytes in its containing directory,

— contains any component in a directory that is not searchable, or

— contains any character in any component that is not valid in its containing directory.

The format of the diagnostic message is not specified, but shall indicate the error detected and the corresponding *pathname* operand.

It shall 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.

4.47.3 Options

The `pathchk` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−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 POSIX.1 {8}2.9.2),

— contains any component longer than {_POSIX_NAME_MAX} bytes, or

— contains any character in any component that is not in the portable filename character set (see 2.2.2.111).

4.47.4 Operands

The following operand shall be supported by the implementation:

pathname A pathname to be checked.

4.47.5 External Influences

4.47.5.1 Standard Input

None.

4.47.5.2 Input Files

None.

4.47.5.3 Environment Variables

The following environment variables shall affect the execution of `pathchk`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.47.5.4 Asynchronous Events

Default.

4.47.6 External Effects

4.47.6.1 Standard Output

None.

4.47.6.2 Standard Error

Used only for diagnostic messages.

4.47.6.3 Output Files

None.

4.47.7 Extended Description

None.

4.47.8 Exit Status

The `pathchk` utility shall exit with one of the following values:

- 0 All *pathname* operands passed all of the checks.
- >0 An error occurred.

4.47.9 Consequences of Errors

Default.

4.47.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

To verify that all pathnames in an imported data interchange archive are legitimate and unambiguous on the current system:

```
pax -f archive | 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 POSIX.1 {8} conforming system 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 \                    1
creation checks.\n" $0 "$path" "$path"                                1
        $reset  # reset the noclobber option in case a trap          1
                # on EXIT depends on it                              1
        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 already exist 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.

History of Decisions Made

The `pathchk` utility is new, commissioned for this standard. It, along with the `set −C` (*noclobber*) option added to the shell, replaces the `mktemp`, `validfnam`, and `create` utilities that appeared in earlier drafts. All of these utilities were attempts to solve a few common problems:

— Verify the validity (for several different definitions of ‘‘valid’’) of a pathname supplied by a user, generated by an application, or imported from an external source,

— Atomically create a file, and

— Perform various string handling functions to generate a temporary file name.

The `test` utility (see 4.62) 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 (see Execution-Time Symbolic Constants for Portability Specification in POSIX.1 {8} 2.9.4) is not in effect. The `pathchk` utility provided here 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 added to the shell (see 3.14.11) can be used to atomically create a file. As with all file creation semantics in POSIX.1 {8}, 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. The `create` utility, included in one earlier draft, provided checking and atomic creation in a single invocation of the utility; these are orthogonal issues and need not be grouped into a single utility. Note that the *noclobber* option also provides a way of creating a lock for process synchronization; since it provides an atomic `create`, there is no race between a test for existence and the following creation if it did not exist.

Having a function like *tmpnam*() in the C Standard {7} is important in many high-level languages. The shell programming language, however, has built-in string manipulation facilities, making it very easy to construct temporary file names. The names needed obviously depend on the application, but are frequently of a form similar to

`$TMPDIR/`*application_abbreviation* `$$.`*suffix*

In cases where there is likely to be contention for a given suffix, a simple shell `for` or `while` loop can be used with the shell *noclobber* option to create a file without risk of collisions, as long as applications trying to use the same filename namespace are cooperating on the use of files after they have been created.

4.48 pax — Portable archive interchange

4.48.1 Synopsis

`pax` [`−cdnv`] [`−f` *archive*] [`−s` *replstr*] . . . [*pattern* . . .] 1

`pax −r` [`−cdiknuv`] [`−f` *archive*] [`−o` *options*] . . . [`−p` *string*] . . . [`−s` *replstr*] . . . [*pattern* . . .] 1 1

`pax −w` [`−dituvX`] [`−b` *blocksize*] [[`−a`] [`−f` *archive*]] [`−o` *options*] . . . [`−s` *replstr*] . . . [`−x` *format*] [*file* . . .] 1

`pax −r −w` [`−diklntuvX`] [`−p` *string*] . . . [`−s` *replstr*] . . . [*file* . . .] *directory*

4.48.2 Description

The `pax` utility shall read, write, and write lists of the members of archive files and copy directory hierarchies. A variety of archive formats shall be supported; see the `−x` *format* option description under 4.48.3.

The action to be taken depends on the presence of the `−r` and `−w` options:

(1) When neither the `−r` option nor the `−w` option is specified, `pax` shall write 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 shall be written out as well.

(2) When the `−r` option is specified, but the `−w` option is not, `pax` shall extract 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 shall be extracted as well. The extracted files shall be 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. 1 1

(3) When the `−w` option is specified and the `−r` option is not, `pax` shall write 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, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

(4) When both the `−r` and `−w` options are specified, `pax` shall copy the file operands to the destination directory.

If no *file* operands are specified, a list of files to copy, one per line, shall be read from the standard input. A file of type directory shall include all of the files in the file hierarchy rooted at the file.

The effect of the copy shall be 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 POSIX.1 {8}, the results are implementation defined; otherwise it shall be 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.

If, when the `-r` option is specified, intermediate directories are necessary to extract an archive member, `pax` shall perform actions equivalent to the POSIX.1 {8} *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` shall write a diagnostic message to standard error for each one that did not match and exit with a nonzero exit status.

The supported archive formats shall be automatically detected on input. The default output archive format shall be implementation defined.

A single archive can span multiple files. The `pax` utility shall determine, in an implementation-defined manner, what file to read or write as the next file.

If the selected archive format supports the specification of linked files, it shall be 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.

4.48.3 Options

The `pax` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that the order of presentation of the `-s` options is significant.

The following options shall be supported by the implementation:

`-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 defined which devices on the system support appending. Additional file formats unspecified by this standard 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 shall be automatically determined on input. Conforming POSIX.2 applications shall 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 (when neither the `-r` option nor the `-w` option is specified, or the `-r` option is specified and the `-w` option is not) or standard output (when the `-w` option is specified and the `-r` option is not).

`-i` Interactively rename files or archive members. For each archive member matching a *pattern* operand or file matching a *file* operand, a prompt shall be written to the file `/dev/tty`. The prompt shall contain the name of the file or archive member, but the format is otherwise unspecified. A line shall then be read from `/dev/tty`. If this line is blank, the file or archive member shall be skipped. If this line consists of a single period, the file or archive member shall be processed with no modification to its name. Otherwise, its name shall be replaced with the contents of the line. The `pax` utility shall immediately exit with a nonzero 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. When both the `-r` and `-w` options are specified, hard links shall 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 shall be matched for each pattern (although members of type directory shall 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 version of this standard does not specify any such options and a Strictly Conforming POSIX.2 Application shall not use the `-o` option.

NOTE: It is expected that future versions of POSIX.2 will offer additional file formats and this option will be used by POSIX.2 and other POSIX standards to specify such features as international file-name and file codeset translations, security, accounting, etc., related to each additional format.

`−p` *string* Specify one or more file characteristic options (privileges). The *string* option-argument shall be a string specifying file characteristics to be retained or discarded on extraction. The string shall consist of the specification characters `a`, `e`, `m`, `o`, and `p`, and/or other, implementation-defined, characters. 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 2.2.2.60), access time, modification time, and any other, implementation-defined, 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-defined file-mode attributes may be preserved.

In the preceding list, ‘‘preserve’’ indicates that an attribute stored in the archive shall be given to the extracted file, subject to the permissions of the invoking process; otherwise, the attribute shall be determined as part of the normal file creation action (see 2.9.1.4).

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` shall 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` shall write a diagnostic message to standard error. Failure to preserve these items shall affect the final exit status, but shall 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 one(s) given last shall take precedence. For example, if `-p eme` is specified, file modification times shall 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 (see 4.20). The concepts of ‘‘address’’ and ‘‘line’’ are meaningless in the context of the `pax` utility, and shall not be supplied. The format shall be:

`-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 shall also be permitted to contain `<newline>` characters.

Any nonnull character can be used as a delimiter (`/` shown here). Multiple `-s` expressions can be specified; the expressions shall be applied in the order specified, terminating with the first successful substitution. The optional trailing `g` shall be as defined in the `ed` utility. The optional trailing `p` shall cause successful substitutions to be written to standard error. File or archive member names that substitute to the empty string shall be 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. If the `-r` option is specified and the `-w` option is not specified, an archive member with the same name as a file in the file system shall be extracted if the archive member is newer than the file. If the `-w` option is specified and the `-r` option is not specified, an archive file member with the same name as a file in the file system shall 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. If both the `-r` and `-w` options are specified, the file in the destination hierarchy shall 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` Produce a verbose table of contents (see 4.48.6.1) if neither the `-r` option nor the `-w` option is specified. Otherwise, list archive member pathnames to standard error (see 4.48.6.2).

`-x` *format* Specify the output archive format. The `pax` utility shall recognize the following formats:

`cpio` The extended `cpio` interchange format specified in
POSIX.1 {8} 10.1.2. The default *blocksize* for this for- 1
mat for character special archive files shall be 5120. 1
Implementations shall support all *blocksize* values 1
less than or equal to 32 256 that are multiples of 1
512.

`ustar` The extended `tar` interchange format specified in
POSIX.1 {8} 10.1.1. The default *blocksize* for this for- 1
mat for character special archive files shall be 1
10 240. Implementations shall support all *blocksize* 1

values less than or equal to 32 256 that are multiples of 512.

Implementation-defined formats shall 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 shall cause `pax` to exit immediately with a nonzero exit status.

`−X` When traversing the file hierarchy specified by a pathname, `pax` shall not descend into directories that have a different device ID [*st_dev* , see POSIX .1 {8}*stat* ()].

The options that operate on the names of files or archive members (`−c`, `−i`, `−n`, `−s`, `−u`, and `−v`) shall interact as follows. When the `−r` option is specified and the `−w` option is not (archive members are being extracted), the archive members shall be ''selected,'' based on the user-specified *pattern* operands as modified by the `−c`, `−n`, and `−u` options. Then, any `−s` and `−i` options shall modify, in that order, the names of the selected files. The `−v` option shall write names resulting from these modifications.

When the `−w` option is specified (files are being archived), the files shall be selected based on the user-specified pathnames as modified by the `−n` and `−u` options. Then, any `−s` and `−i` options shall, in that order, modify the names of these selected files. The `−v` option shall write names resulting from these modifications.

If both the `−u` and `−n` options are specified, `pax` shall not consider a file selected unless it is newer than the file to which it is compared.

4.48.4 Operands

The following operands shall be supported by the implementation:

directory The destination directory pathname for copies when both the `−r` and `−w` options are specified.

file A pathname of a file to be copied or archived.

pattern A pattern matching one or more pathnames of archive members. A pattern shall be given in the name-generating notation of the pattern matching notation in 3.13, including the filename expansion rules in 3.13.3. The default, if no *pattern* is specified, is to select all members in the archive.

4.48.5 External Influences

4.48.5.1 Standard Input

If the `-w` option is specified, the standard input shall be used only if no *file* operands are specified. It shall be a text file containing a list of pathnames, one per line, without leading or trailing `<blank>`s.

If neither the `-f` nor `-w` options are specified, the standard input shall be an archive file. (See 4.48.5.2.)

Otherwise, the standard input shall not be used.

4.48.5.2 Input Files

The input file named by the *archive* option-argument, or standard input when the archive is read from there, shall be a file formatted according to one of the specifications in POSIX.1 {8} 10.1, or some other, implementation-defined, format.

The file `/dev/tty` shall be used to write prompts and read responses.

4.48.5.3 Environment Variables

The following environment variables shall affect the execution of `pax`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter 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 This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and the behavior of character classes within regular expressions and pattern matching.

LC_MESSAGES This variable shall determine the processing of affirmative responses and the language in which messages should be written.

LC_TIME This variable shall determine the format and contents of date and time strings when the `−v` option is specified.

4.48.5.4 Asynchronous Events

Default.

4.48.6 External Effects

4.48.6.1 Standard Output

If the `−w` option is specified and neither the `−f` nor `−r` options are specified, the standard output shall be the archive formatted according to one of the specifications in POSIX.1 {8} 10.1, or some other implementation-defined format. (See `−x` *format* under 4.48.3.)

If neither the `−r` option nor the `−w` option is specified, the table of contents of the selected archive members shall be written to standard output using the following format: 1

`"%s\n"`, <*pathname*>

If neither the `−r` option nor the `−w` option is specified, but the `−v` option is specified, the table of contents of the selected archive members shall be written to standard output using the following formats:

For pathnames representing hard links to previous members of the archive:

`"%sΔ==Δ%s\n"`, <*ls −l listing*> , <*linkname*>

For all other pathnames:

`"%s\n"`, <*ls −l listing*>

where <*ls −l listing*> shall be the format specified by the `ls` utility (see 4.39) 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>`-separated fields shall be written.

When writing a table of contents of selected archive members, standard output shall not be buffered more than a line at a time.

4.48.6.2 Standard Error

If either or both of the `−r` option and the `−w` option are specified as well as the `−v` option, `pax` shall write the pathnames it processes to the standard error output using the following format: 1

`"%s\n"`, *<pathname>*

These pathnames shall be written as soon as processing is begun on the file or archive member, and shall be flushed to standard error. The trailing `<newline>`, which shall not be buffered, shall 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 shall be written to standard error in the following format:

`"%s∆>>∆%s\n"`, *<original pathname>* , *<new pathname>* 2

In all operating modes of `pax` (see 4.48.2), 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 nonprintable characters in pathnames or linknames are written.

4.48.6.3 Output Files

If the `−r` option is specified, the extracted or copied output files shall be of the archived file type.

If the `−w` option is specified, but the `−r` option is not, the output file named by the `−f` option argument shall be a file formatted according to one of the specifications in POSIX.1 {8}10.1, or some other, implementation-defined, format.

4.48.7 Extended Description

None.

4.48.8 Exit Status

The `pax` utility shall exit with one of the following values:

0 All files were processed successfully.

>0 An error occurred.

4.48.9 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 shall be written to standard error and a nonzero exit status shall be returned, but processing shall continue. In the case where `pax` cannot create a link to a file, `pax` shall 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.

4.48.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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.

The `−p` (privileges) option was invented to reconcile differences between historical 1
`tar` and `cpio` implementations. In particular, the two utilities used `−m` in 1
diametrically opposed ways. The `−p` option also provides a consistent means of 1
extending the ways in which future file attributes can be addressed, such as for 1
enhanced security systems or high-performance files. Although it may seem com- 1
plex, there are really two modes that will be most commonly used: 1

`-p e` ‘‘Preserve everything.’’ This would be used by the historical super- 1
user, someone with all the appropriate privileges, to preserve all 1
aspects of the files as they are recorded in the archive. The `e` flag is 1
the sum of `o` and `p`, and other implementation-defined attributes. 1

`-p p` ‘‘Preserve’’ the file mode bits. This would be used by the user with 1
regular privileges who wished to preserve aspects of the file other 1
than the ownership. The file times are preserved by default, but two 1
other flags are offered to disable these and use the time of extraction. 1

History of Decisions Made

The description of `pax` was adopted from a command written by Glenn Fowler of AT&T. It is a new utility, commissioned for this standard.

The table of contents output is written to standard output to facilitate pipeline processing.

The output archive formats required are those defined in POSIX.1 {8}; others, such as the historical `tar` format, may be added as an extension.

The one pathname per line format of standard input precludes pathnames containing <newline>s. 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.

An earlier draft had hard links displaying for all pathnames. This was removed 1
because it complicates the output of the non −`v` case and does not match historical 1
`cpio` usage. The hard-link information is available in the −`v` display. 1

The working group realizes that the presence of symbolic links will affect certain `pax` operations. Historical practice, in both System V and BSD-based systems, is that the physical traversal of the file hierarchy shall be the default, and an option is provided to cause the utility to do a logical traversal, that is, follow symbolic links. Historical practice has not been so consistent as to what option is used to cause the logical traversal; BSD systems have used −`h` (`cp` and `tar`) and −`L` (`ls`), while the *SVID* specifies −`L` (`cpio` and `ls`). Given this inconsistency, the −`L` option is recommended.

The archive formats described in POSIX.1 {8} have certain restrictions that have been brought along from historical usage. For example, there are restrictions on the length of pathnames stored in the archive. When `pax` is used in −`rw` mode, copying directory hierarchies, there is no stated dependency on these archive formats. Therefore, such restrictions should not apply.

The POSIX.2 working group is currently devising a new archive format to be pub- 1
lished in a revision or amendment to this standard. It is expected that the `ustar` 1
and `cpio` formats then will be retired from a future version of POSIX.1 {8}. This 1
new format will address all restrictions and new requirements for security label-
ing, etc. The `pax` utility should be upward-compatible enough to handle any such
changes. The reason that the default −`x` *format* output format is implementation
defined is to reserve the default format for this new standard interface. The −`o` 1
option was devised to provide means of controlling the many aspects of interna- 1
tional and security concerns without expending the entire alphabet of option 1
letters for this, and possibly other, file formats. The −`o` string is meant to be 1
specific for each −`x` format. Control of various file permissions and attributes that 1
can be expressed in a binary way will continue to use the −`p` (permissions) option; 1
the −`o` will be reserved for more involved requirements and will probably take a 1

```
    pax -o name=value,name=value -o name=value                  1
```

approach. 1

The fundamental difference in how `cpio` and `tar` viewed the world was in the way directories were treated. The `cpio` utility did not treat directories differently from other files, and to select a directory and its contents required that each file in the hierarchy be explicitly specified. For `tar`, a directory matched every file in the file hierarchy it rooted.

The `pax` utility offers both interfaces; by default, directories map into the file hierarchy they root. The `-d` option causes `pax` to skip any file not explicitly referenced, as `cpio` traditionally did. The `tar`-style behavior was chosen as the default because it was believed that this was the more common usage, and because `tar` is the more commonly available interface, as it was historically provided on both System V and BSD implementations. Because a file may be matched more than once without causing it to be selected multiple times, the traditional usage of piping an `ls` or `find` to the archive command works as always.

The Data Interchange Format specification of POSIX.1 {8} requires that processes with ''appropriate privileges'' shall always restore the ownership and permissions of extracted files exactly as archived. If viewed from the historic equivalence between super-user and ''appropriate privileges,'' there are two problems with this requirement. First, users running as super-users may unknowingly set dangerous permissions on extracted files. Second, it is needlessly limiting in that super-users cannot extract files and own them as super-user unless the archive was created by the super-user. (It should be noted that restoration of ownerships and permissions for the super-user, by default, is historical practice in `cpio`, but not in `tar`.) In order to avoid these two problems, the `pax` specification has an additional ''privilege'' mechanism, the `-p` option. Only a `pax` invocation with the POSIX.1 {8} privileges needed, and which has the `-p` option set using the `e` specification character, has the ''appropriate privilege'' to restore full ownership and permission information.

Note also that POSIX.1 {8} 10.1 requires that the file ownership and access permissions shall be set, on extraction, in the same fashion as the POSIX.1 {8} *creat*() function when provided the mode stored in the archive. This means that the file creation mask of the user is applied to the file permissions.

The default *blocksize* value of 5120 for `cpio` was selected because it is one of the standard block-size values for `cpio`, set when the `-B` option is specified. (The other default block-size value for `cpio` is 512, and this was felt to be too small.) The default block value of 10 240 for `tar` was selected as that is the standard block-size value for BSD `tar`. The maximum block size of 32 256 (2^{15}–512) is the largest multiple of 512 that fits into a signed 16-bit tape controller transfer register. There are known limitations in some historic system that would prevent larger blocks from being accepted. Historic values were chosen to make compatibility with existing scripts using `dd` or similar utilities to manipulate archives more likely. Also, default block sizes for any file type other than character special has been deleted from the standard as unimportant and not likely to affect the structure of the resulting archive.

Implementations are permitted to modify the block-size value based on the archive format or the device to which the archive is being written. This is to provide implementations the opportunity to take advantage of special types of devices, and should not be used without a great deal of consideration as it will almost certainly decrease archive portability.

The −n option in early drafts had three effects; the first was to cause special characters in patterns to not be treated specially. The second was to cause only the first file that matched a pattern to be extracted. The third was to cause pax to write a diagnostic message to standard error when no file was found matching a specified pattern. Only the second behavior is retained by POSIX.2, for many reasons. First, it is in general a bad idea for a single option to have multiple effects. Second, the ability to make pattern matching characters act as normal characters is useful for other parts of pax than just file extraction. Third, a finer degree of control over the special characters is useful, because users may wish to normalize only a single special character in a single file name. Fourth, given a more general escape mechanism, the previous behavior of the −n option can be easily obtained using the −s option or a sed script. Finally, writing a diagnostic message when a pattern specified by the user is unmatched by any file is useful behavior in all cases.

There are two methods of copying subtrees in POSIX.2. The other method is described as part of the cp utility (see 4.13). Both methods are historical practice: cp provides a simpler, more intuitive interface, while pax offers a finer granularity of control. Each provides additional functionality to the other; in particular, pax maintains the hard-link structure of the hierarchy, while cp does not. It is the intention of the working group that the results be similar (using appropriate option combinations in both utilities). The results are not required to be identical; there seemed insufficient gain to applications to balance the difficulty of implementations having to guarantee that the results would be exactly identical.

A single archive may span more than one file. See POSIX.1 {8} 10.1.3. While POSIX.1 {8} only refers to reading the archive file, it is reasonable that the format utility may also determine, in an implementation-defined manner, the next file to write. It is suggested that implementations provide informative messages to the user on the standard error whenever the archive file is changed.

The −d option (do not create intermediate directories not listed in the archive) found in previous drafts of this standard was originally provided as a complement to the historic −d option of cpio. It has been deleted.

The −s option in earlier drafts specified a subset of the substitution command from the ed utility. As there was no reason for only a subset to be supported, the −s option is now compatible with the current ed specification. Since the delimiter can be any nonnull character, the following usage with single spaces is valid:

```
pax -s " foo bar " ...
```

The −t option (specify an implementation-defined identifier naming an input or output device) found in earlier drafts has been deleted because it is not historical practice and of limited utility. In particular, historic versions of neither cpio nor tar had the concept of devices that were not mapped into the file system; if the devices are mapped into the file system, the −f option is sufficient.

The −o and −p options found in previous versions of this standard have been renamed to be −p and −t, respectively, to correspond more closely with the historic tar and cp utilities.

The default behavior of `pax` with regard to file modification times is the same as historical implementations of `tar`. It is not the historical behavior of `cpio`.

Because the `−i` option uses `/dev/tty`, utilities without a controlling terminal will not be able to use this option.

The `−y` option, found in earlier drafts, has been deleted because a line containing a single period for the `−i` option has equivalent functionality. The special lines for the `−i` option (a single period and the empty line) are historical practice in `cpio`.

In earlier drafts, an `−e` *charmap* option was included to increase portability of 1
files between systems using different coded character sets. This option was omit- 1
ted because it was apparent that consensus could not be formed for it. It was an 1
interface without implementation experience and overloaded the charmap file 1
concept to provide additional uses its original authors had not intended. The 1
developers of POSIX.2 will consider other mechanisms for transporting files with 1
nonportable names as they develop the new interchange format, described earlier. 1

The `−k` option was added to address international concerns about the dangers involved in the character set transformations of `−e` (if the target character set were different than the source, the file names might be transformed into names matching existing files) and was made more general to also protect files transferred between file systems with different {NAME_MAX} values (truncating a filename on a smaller system might also inadvertently overwrite existing files). As stated, it prevents any overwriting, even if the target file is older than the source, which is seen as a generally useful feature anyway.

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.

Some of the file characteristics referenced in this specification may not be supported by some archive formats. For example, neither the `tar` nor `cpio` formats contain the file access time. For this reason, the `e` specification character has been provided, intended to cause all file characteristics specified in the archive to be retained.

It is required that extracted directories, by default, have their access and modification times and permissions set to the values specified in the archive. This has obvious problems in that the directories are almost certainly modified after being extracted and that directory permissions may not permit file creation. One possible solution is to create directories with the mode specified in the archive, as modified by the *umask* of the user, plus sufficient permissions to allow file creation. After all files have been extracted, `pax` would then reset the access and modification times and permissions as necessary.

When the `−r` option is specified, and the `−w` option is not, implementations are permitted to overwrite files when the archive has multiple members with the same name. This may fail, of course, if permissions on the first version of the file do not permit it to be overwritten.

4.49 `pr` — Print files

4.49.1 Synopsis

`pr` [`+`*page*] [`−`*column*] [`−adFmrt`] [`−e`[*char*][*gap*]] [`−h` *header*] [`−i`[*char*][*gap*]]
[`−l` *lines*] [`−n`[*char*][*width*]] [`−o` *offset*] [`−s`[*char*]] [`−w` *width*] [*file* . . .]

4.49.2 Description

The `pr` utility is a printing and pagination filter. If multiple input files are specified, each shall be read, formatted, and written to standard output. By default, the input shall be separated into 66-line pages, each with:

— A 5-line header that includes the page number, date, time, and the path- 1
name of the file. 1

— A 5-line trailer consisting of blank lines. 1

If standard output is associated with a terminal, diagnostic messages shall be deferred until the `pr` utility has completed processing.

When options specifying multicolumn output are specified, output text columns shall be of equal width; input lines that do not fit into a text column shall be truncated. By default, text columns shall be separated with at least one `<blank>`.

4.49.3 Options

The `pr` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that: the *page* option has a `'+'` delimiter; *page* and *column* can be multidigit 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 shall be supported by the implementation. In the following
option descriptions, *column*, *lines*, *offset*, *page*, and *width* are positive decimal 1
integers; *gap* is a nonnegative decimal integer. 1

`+`*page* Begin output at page number *page* of the formatted input.

`−`*column* Produce output that is *columns* wide (default shall be 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` shall be assumed for multiple text-column output. Whether or not text columns are balanced is unspecified, but a text column shall 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 (e.g., 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, etc.).

`−d` Produce output that is double-spaced; append an extra `<newline>` following every `<newline>` found in the input.

`−e`[*char*][*gap*]
Expand each input `<tab>` 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 shall default to 8. All `<tab>` characters in the input shall be expanded into the appropriate number of `<space>`s. If any nondigit character, *char*, is specified, it shall be used as the input tab character.

`−F` Use a `<form-feed>` character for new pages, instead of the default behavior 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. See 4.49.6.1.

`−i`[*char*][*gap*]
In output, replace multiple `<space>`s with `<tab>`s wherever two or more adjacent `<space>`s reach column positions $gap+1$, $2*gap+1$, $3*gap+1$, etc. If *gap* is zero or is omitted, default `<tab>` settings at every eighth column position shall be assumed. If any nondigit character, *char*, is specified, it shall 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 shall suppress both the header and trailer, as if the `−t` option were in effect.

`−m` Merge files. Standard output shall 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 shall support merging of at least nine *file* operands.

`−n`[*char*][*width*]
Provide *width*-digit line numbering (default for *width* shall be 5). The number shall occupy the first *width* column positions of each text column of default output or each line of `−m` output. If *char* (any nondigit character) is given, it shall be appended to the line number to separate it from whatever follows (default for *char* shall be a `<tab>`).

−o *offset* Each line of output shall be preceded by offset <space>s. If the −o option is not specified, the default offset shall be zero. The space taken shall be in addition to the output line width (see −w option below).

−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>s (default for *char* shall be 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 shall be 72. If the −w option is not specified and the −s option is specified, the default width shall be 512.

For single column output, input lines shall not be truncated.

4.49.4 Operands

The following operand shall be supported by the implementation:

file A pathname of a file to be written. If no *file* operands are specified, or if a *file* operand is −, the standard input shall be used.

4.49.5 External Influences

4.49.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is −. See Input Files.

4.49.5.2 Input Files

The input files shall be text files.

4.49.5.3 Environment Variables

The following environment variables shall affect the execution of pr:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and which characters are defined as printable (character class `print`). Nonprintable characters still shall be written to standard output, but shall be not counted for the purpose for column-width and line-length calculations.

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

LC_TIME This variable shall determine the format of the date and time for use in writing header lines.

TZ This variable shall determine the time zone for use in writing header lines.

4.49.5.4 Asynchronous Events

If `pr` receives an interrupt while writing to a terminal, it shall flush all accumulated error messages to the screen before terminating.

4.49.6 External Effects

4.49.6.1 Standard Output

The `pr` utility output shall be a paginated version of the original file (or files). This pagination shall be accomplished using either `<form-feed>`s or a sequence of `<newline>`s, as controlled by the `−F` option. Page headers shall be generated unless the `−t` option is specified. The page headers shall 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), shall be 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>`, 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 shall be used, but the time used shall 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 shall be replaced by a null string.

If the `-h` option is specified, the *file* field shall be replaced by the *header* argument.

4.49.6.2 Standard Error

Used only for diagnostic messages.

4.49.6.3 Output Files

None.

4.49.7 Extended Description

None.

4.49.8 Exit Status

The `pr` utility shall exit with one of the following values:

- 0 All files were written successfully.
- >0 An error occurred.

4.49.9 Consequences of Errors

Default.

4.49.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

To print a numbered list of all files in the current directory:

```
ls -a | pr -n -h "Files in $(pwd)."
```

History of Decisions Made

This utility is one of those that does not follow the Utility Syntax Guidelines because of its historical origins. The working group could have added new options that obeyed the guidelines (and marked the old options *obsolescent*) or devised an entirely new utility; there are examples of both actions in this standard. For this utility, it chose to leave some of the options as they are because of their heavy 1
usage by existing applications. However, due to interest in the international

community, the developers of the standard have agreed to provide an alternative syntax for the next version of this standard that conforms to the spirit of the Utility Syntax Guidelines. This new syntax will be accompanied by the existing syntax, marked as obsolescent. System implementors are encouraged to develop and promulgate a new syntax for `pr`, perhaps using a different utility name, that can be adopted for the next version of this standard.

Implementations are required to accept option arguments to the `−h`, `−l`, `−o`, and `−w` options whether presented as part of the same argument or as a separate argument to `pr`, as suggested by the utility syntax guidelines. The `−n` and `−s` options, however, are specified as in historical practice because they are frequently specified without their optional arguments. If a `<blank>` were allowed before the option-argument in these cases, a file operand could mistakenly be interpreted as an option-argument in historical applications.

Historical implementations of the `pr` utility have differed in the action taken for the `−f` option. BSD uses it as described here for the `−F` option; System V uses it to change trailing `<newline>`s on each page to a `<form-feed>` and, if standard output is a TTY device, sends an `<alert>` to standard error and reads a line from `/dev/tty` before the first page. Draft 9 incorrectly specified part of the System V behavior, raising several ballot objections. There were strong arguments from both sides of this issue concerning existing practice and additional arguments against the System V `−f` behavior, on the grounds that it was not a modular design to have the behavior of an option change depending on where output is directed. Therefore, the `−f` option is not specified and the `−F` option has been added.

The `−p` option was omitted since it represents a purely interactive usage. 1

The *<output of date>* field in the `−l` format is specified only for the POSIX Locale. As noted, the format can be different in other locales. No mechanism for defining this is present in this standard, as the appropriate vehicle is a messaging system; i.e., the format should be specified as a ‘‘message.’’

4.50 `printf` — Write formatted output

4.50.1 Synopsis

`printf` *format* [*argument* ...]

4.50.2 Description

The `printf` utility shall write formatted operands to the standard output. The *argument* operands shall be formatted under control of the *format* operand.

4.50.3 Options

None.

4.50.4 Operands

The following operands shall be supported by the implementation:

format A string describing the format to use to write the remaining operands; see 4.50.7.

argument The strings to be written to standard output, under the control of *format* ; see 4.50.7.

4.50.5 External Influences

4.50.5.1 Standard Input

None.

4.50.5.2 Input Files

None.

4.50.5.3 Environment Variables

The following environment variables shall affect the execution of `printf`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

LC_NUMERIC This variable shall determine the locale for numeric formatting. It shall affect the format of numbers written using the `e`, `E`, `f`, `g`, and `G` conversion characters (if supported).

4.50.5.4 Asynchronous Events

Default.

4.50.6 External Effects

4.50.6.1 Standard Output

See 4.50.7.

4.50.6.2 Standard Error

Used only for diagnostic messages.

4.50.6.3 Output Files

None.

4.50.7 Extended Description

The *format* operand shall be used as the *format* string described in 2.12 with the following exceptions:

(1) A <space> character in the format string, in any context other than a flag of a conversion specification, shall be treated as an ordinary character that is copied to the output.

(2) A Δ character in the format string shall be treated as a Δ character, not as a <space>.

(3) In addition to the escape sequences shown in Table 2-15 (see 2.12), *ddd*, where *ddd* is a one-, two-, or three-digit octal number, shall be written as a byte with the numeric value specified by the octal number.

(4) The implementation shall not precede or follow output from the `d` or `u` conversion specifications with `<blank>`s not specified by the *format* operand.

(5) The implementation shall not precede output from the `o` conversion specification with zeroes not specified by the *format* operand.

(6) The `e`, `E`, `f`, `g`, and `G` conversion specifications need not be supported.

(7) An additional conversion character, `b`, shall be supported as follows. The argument shall be taken to be a string that may contain backslash-escape sequences. The following backslash-escape sequences shall be supported:

(a) The escape sequences listed in Table 2-15, which shall be converted to the characters they represent;

(b) `\0`*ddd* , where *ddd* is a zero-, one-, two-, or three-digit octal number that shall be converted to a byte with the numeric value specified by the octal number;

(c) `\c`, which shall not be written and shall 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 shall 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 shall be taken to be infinite, so all bytes up to the end of the converted string shall be written.

(8) For each specification that consumes an argument, the next argument operand shall be evaluated and converted to the appropriate type for the conversion as specified below.

(9) The *format* operand shall be reused as often as necessary to satisfy the argument operands. Any extra `c` or `s` conversion specifications shall be evaluated as if a null string argument were supplied; other extra conversion specifications shall 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.

(10) If a character sequence in the *format* operand begins with a `%` character, but does not form a valid conversion specification, the behavior is unspecified.

The *argument* operands shall be treated as strings if the corresponding conversion character is `b`, `c`, or `s`; otherwise, it shall be evaluated as a C constant, as described by the C Standard {7}, with the following extensions:

— A leading plus or minus sign shall be allowed.

— If the leading character is a single- or double-quote, the value shall be the numeric value in the underlying code set 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 shall be written to standard error and the utility shall not exit with a zero exit status, but shall continue processing any remaining operands and shall write the value accumulated at the time the error was detected to standard output.

4.50.8 Exit Status

The `printf` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.50.9 Consequences of Errors

Default.

4.50.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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 right, and print them out. The numbers are right-justified and separated by a single `<tab>`. 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 C Standard {7} 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 ISO/IEC 646 {1} as the underlying code set, 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 ISO/IEC 646 {1} code set
`43`	Numeric value of the character ‘‘+’’ in ISO/IEC 646 {1} code set
`45`	Numeric value of the character ‘‘−’’ in ISO/IEC 646 {1} code set

Note that in a locale with multibyte 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 C Standard {7}.

History of Decisions Made

The `printf` utility was added to provide functionality that has historically been provided by `echo`. However, due to irreconcilable differences in the various versions of `echo` extant, the version in this standard has few special features, leaving those to this new `printf` utility, which is based on one in the Ninth Edition at AT&T Bell Labs.

The Extended Description almost exactly matches the C Standard {7} *printf* () function, although it is described in terms of the file format notation in 2.12.

The floating point formatting conversion specifications 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 doesn't 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 C Standard {7}*printf* () function on which it is based, makes no special provision for dealing with multibyte 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 multibyte 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 C Standard {7} are not recognized 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 nonhex-digit character. In the C Standard {7}, 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 C Standard {7}; it has been added here as a portable way to process backslash-escapes expanded in string operands as provided by the System V version of the `echo` utility. See also the rationale for `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. If written in C, the `printf` utility could use the *strtol* () function to parse optionally signed numeric arguments, *strtoul* () to parse unsigned numeric arguments, and *strtod* () to parse floating point arguments (if floating point conversions are supported). 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.

4.51 pwd — Return working directory name

4.51.1 Synopsis

```
pwd
```

4.51.2 Description

The pwd utility shall write an absolute pathname of the current working directory to standard output.

4.51.3 Options

None.

4.51.4 Operands

None.

4.51.5 External Influences

4.51.5.1 Standard Input

None.

4.51.5.2 Input Files

None.

4.51.5.3 Environment Variables

The following environment variables shall affect the execution of pwd:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

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

4.51.5.4 Asynchronous Events

Default.

4.51.6 External Effects

4.51.6.1 Standard Output

The `pwd` utility output shall be an absolute pathname of the current working directory:

`"%s\n"`, <*directory pathname*>

4.51.6.2 Standard Error

Used only for diagnostic messages.

4.51.6.3 Output Files

None.

4.51.7 Extended Description

None.

4.51.8 Exit Status

The `pwd` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.51.9 Consequences of Errors

If an error is detected, output shall not be written to standard output, a diagnostic message shall be written to standard error, and the exit status shall not be zero.

4.51.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Some implementations have historically provided `pwd` as a shell special built-in command.

History of Decisions Made

In most utilities, if an error occurs, partial output may be written to standard output. This does not happen in historical implementations of `pwd`. Because `pwd` is frequently used in existing shell scripts without checking the exit status, it is important that the historical behavior is required here; therefore, the Consequences of Errors subclause specifically disallows any partial output being written to standard output.

4.52 `read` — Read a line from standard input

4.52.1 Synopsis

`read [−r]` *var* ...

4.52.2 Description

The `read` utility shall read a single line from standard input.

By default, unless the `−r` option is specified, backslash (\) shall act as an escape character, as described in 3.2.1.

The line shall be split into fields (see the definition in 3.1.3) as in the shell (see 3.6.5); the first field shall be assigned to the first variable *var,* the second field to the second variable *var,* etc. If there are fewer *var* operands specified than there are fields, the leftover fields and their intervening separators shall be assigned to the last *var*. If there are fewer fields than *var* s, the remaining *var* s shall be set to empty strings.

The setting of variables specified by the *var* operands shall affect the current shell execution environment; see 3.12.

4.52.3 Options

The `read` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−r` Do not treat a backslash character in any special way. Consider each backslash to be part of the input line.

4.52.4 Operands

The following operands shall be supported by the implementation:

var The name of an existing or nonexisting shell variable.

4.52.5 External Influences

4.52.5.1 Standard Input

The standard input shall be a text file.

4.52.5.2 Input Files

None.

4.52.5.3 Environment Variables

The following environment variables shall affect the execution of `read`:

IFS This variable shall determine the internal field separators used to delimit fields. See 3.5.3.

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**. 2.6.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.52.5.4 Asynchronous Events

Default.

4.52.6 External Effects

4.52.6.1 Standard Output

None.

4.52.6.2 Standard Error

Used only for diagnostic messages.

4.52.6.3 Output Files

None.

4.52.7 Extended Description

None.

4.52.8 Exit Status

The `read` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 End-of-file was detected or an error occurred.

4.52.9 Consequences of Errors

Default.

4.52.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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.

The text in 2.11.5.2 indicates that 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.

Since `read` 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:

```
(read foo) 1
nohup read ... 1
find . -exec read ... \; 1
```

it will not affect the shell variables in the caller's environment. 1

History of Decisions Made

The `read` utility has historically been a shell built-in. It was separated off into its own clause to take advantage of the standard's richer description of functionality at the utility level.

The `−r` option was added to enable `read` to subsume the purpose of the historical `line` utility.

4.53 `rm` — Remove directory entries

4.53.1 Synopsis

`rm [−fiRr]` *file* . . .

4.53.2 Description

The `rm` utility shall remove 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 (i.e., the final pathname component), `rm` shall write a diagnostic message to standard error and do nothing more with such operands.

For each *file* the following steps shall be 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 shall be 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)] shall be 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` shall perform actions equivalent to the POSIX.1 {8} *rmdir*() function called with a pathname of the current file used as the *path* argument. If the current file is not a directory, `rm` shall perform actions equivalent to the POSIX.1 {8} *unlink*() function called with a pathname of the current file used as the *path* argument.

If this fails for any reason, `rm` shall write a diagnostic message to standard error, do nothing more with the current file, and go on to any remaining files.

The `rm` utility shall be able to descend to arbitrary depths in a file hierarchy, and shall not fail due to path length limitations (unless an operand specified by the user exceeds system limitations).

4.53.3 Options

The `rm` utility shall conform to the utility argument syntax guidelines described 2
in 2.10.2. 2

The following options shall be supported by the implementation:

`−f` Do not prompt for confirmation. Do not write diagnostic messages or modify the exit status in the case of nonexistent operands. Any previous occurrences of the `−i` option shall be ignored.

`−i` Prompt for confirmation as described in 4.53.2. Any previous occurrences of the `−f` option shall be ignored.

`−R` Remove file hierarchies. See 4.53.2.

`−r` Equivalent to `−R`.

4.53.4 Operands

The following operand shall be supported by the implementation:

file A pathname of a directory entry to be removed.

4.53.5 External Influences

4.53.5.1 Standard Input

Used to read an input line in response to each prompt specified in 4.53.6.1. Otherwise, the standard input shall not be used.

4.53.5.2 Input Files

None.

4.53.5.3 Environment Variables

The following environment variables shall affect the execution of `rm`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements used in the extended regular expression defined for the `yesexpr` locale keyword in the LC_MESSAGES category.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior 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 This variable shall determine the processing of affirmative responses and the language in which messages should be written.

4.53.5.4 Asynchronous Events

Default.

4.53.6 External Effects

4.53.6.1 Standard Output

None.

4.53.6.2 Standard Error

Prompts shall be written to standard error under the conditions specified in 4.53.2 and 4.53.3. The prompts shall contain the *file* pathname, but their format is otherwise unspecified. The standard error shall also be used for diagnostic messages.

4.53.6.3 Output Files

None.

4.53.7 Extended Description

None.

4.53.8 Exit Status

The `rm` utility shall exit with one of the following values:

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.

4.53.9 Consequences of Errors

Default.

4.53.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The *SVID* requires that systems do not permit the removal of the last link to an executable binary file that is being executed. 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. 1

For absolute clarity, paragraphs (2)(b) and (3) in 4.53.2, describing `rm`'s behavior when prompting for confirmation, should be interpreted in the following manner:

```
if ((NOT f_option) AND
        ((not_writable AND input_is_terminal) OR i_option))
```

It is forbidden to remove the names dot and dot-dot in order to avoid the consequences of inadvertently doing something like:

```
rm −r .*
```

The following command

```
rm a.out core
```

removes the directory entries `a.out` and `core`.

The following command

```
rm -Rf junk
```

removes the directory `junk` and all its contents, without prompting.

History of Decisions Made

The exact format of the interactive prompts is unspecified. Only the general nature of the contents of prompts are specified, because implementations may desire more descriptive prompts than those used on historical implementations. Therefore, an application not using the `−f` option, or using the `−i` option relies on the system to provide the most suitable dialogue directly with the user, based on the behavior specified.

The `−r` option is existing practice on all known systems. The synonym `−R` option is provided for consistency with the other utilities in this standard that provide options requesting recursive descent.

The behavior of the `−f` option in historical versions of `rm` is inconsistent. In general, along with ‘‘forcing’’ the unlink without prompting for permission, it always causes diagnostic messages to be suppressed and the exit status to be unmodified for nonexistent operands and files that cannot be unlinked. In some versions, however, the `−f` option suppresses usage messages and system errors as well. Suppressing such messages is not a service to either shell scripts or users.

It is less clear that error messages regarding unlinkable files should be suppressed. Although this is historical practice, this standard does not permit the `−f` option to suppress such messages.

When given the `−r` and `−i` options, historical versions of `rm` prompt the user twice for each directory, once before removing its contents and once before actually attempting to delete the directory entry that names it. This allows the user to ‘‘prune’’ the file hierarchy walk. Historical versions of `rm` were inconsistent in that some did not do the former prompt for directories named on the command line and others had obscure prompting behavior when the `−i` option was specified and the permissions of the file did not permit writing. The POSIX.2 `rm` differs little from historic practice, but does require that prompts be consistent. Historical versions of `rm` were also inconsistent in that prompts were done to both standard output and standard error. POSIX.2 requires that prompts be done to standard error, for consistency with `cp` and `mv` and to allow existing extensions to `rm` that provide an option to list deleted files on standard output.

The `rm` utility is required to descend to arbitrary depths so that any file hierarchy may be deleted. This means, for example, that the `rm` utility cannot run out of file descriptors during its descent, i.e., if the number of file descriptors is limited, `rm` cannot be implemented in the historical fashion where a file descriptor is used per directory level. Also, `rm` is not permitted to fail because of path length restrictions, unless an operand specified by the user is longer than {PATH_MAX}.

4.54 `rmdir` — Remove directories

4.54.1 Synopsis

`rmdir [−p]` *dir* . . .

4.54.2 Description

The `rmdir` utility shall remove the directory entry specified by each *dir* operand, which shall refer to an empty directory.

Directories shall 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 shall be specified before the parent directory so that the parent directory will be empty when the `rmdir` utility tries to remove it.

4.54.3 Options

The `rmdir` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−p` Remove all directories in a pathname. For each *dir* operand:

(1) The directory entry it names shall be removed.

(2) If the *dir* operand includes more than one pathname component, effects equivalent to the following command shall occur:

`rmdir −p $(dirname` *dir* `)`

4.54.4 Operands

The following operand shall be supported by the implementation:

dir A pathname of an empty directory to be removed.

4.54.5 External Influences

4.54.5.1 Standard Input

None.

4.54.5.2 Input Files

None.

4.54.5.3 Environment Variables

The following environment variables shall affect the execution of `rmdir`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.54.5.4 Asynchronous Events

Default.

4.54.6 External Effects

4.54.6.1 Standard Output

None.

4.54.6.2 Standard Error

Used only for diagnostic messages.

4.54.6.3 Output Files

None.

4.54.7 Extended Description

None.

4.54.8 Exit Status

The `rmdir` utility shall exit with one of the following values:

0 Each directory entry specified by a *dir* operand was removed successfully.

>0 An error occurred.

4.54.9 Consequences of Errors

Default.

4.54.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

On historical System V systems, the `−p` option also caused a message to be written to the standard output. The message indicated whether the whole path was removed or part of the path remains for some reason. The Standard Error subclause requires this diagnostic when the entire path specified by a *dir* operand is not removed, but does not allow the status message reporting success to be written as a diagnostic.

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.

The `rmdir` utility on System V also included an `−s` option that suppressed the informational message output by the `−p` option. This option has been omitted because the informational message is not specified by POSIX. 2.

4.55 `sed` — Stream editor

4.55.1 Synopsis

`sed [−n]` *script* [*file* . . .]

`sed [−n][−e` *script*] . . . [`−f` *script_file*] . . . [*file* . . .]

4.55.2 Description

The `sed` utility is a stream editor that shall read one or more text files, make editing changes according to a script of editing commands, and write the results to standard output. The script shall be obtained from either the *script* operand string or a combination of the option-arguments from the `−e` *script* and `−f` *script_file* options.

4.55.3 Options

The `sed` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that the order of presentation of the `−e` and `−f` options is significant.

The following options shall be supported by the implementation:

`−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 shall have the same properties as the *script* operand, described in 4.55.4.

`−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 shall be written.

Multiple `−e` and `−f` options may be specified. All commands shall be added to the script in the order specified, regardless of their origin.

4.55.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a file whose contents shall be read and edited. If multiple *file* operands are specified, the named files shall be read in the order specified and the concatenation shall be edited. If no *file* operands are specified, the standard input shall be used.

script A string to be used as the script of editing commands. The application shall not present a *script* that violates the restrictions of a text file (see 2.2.2.151), except that the final character need not be a `<newline>`.

4.55.5 External Influences

4.55.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.55.5.2 Input Files

The input files shall be text files. The *script_file* s named by the `−f` option shall consist of editing commands, one per line.

4.55.5.3 Environment Variables

The following environment variables shall affect the execution of `sed`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), and the behavior of character classes within regular expressions.

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

4.55.5.4 Asynchronous Events

Default.

4.55.6 External Effects

4.55.6.1 Standard Output

The input files shall be 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 shall be written to standard output.

4.55.6.2 Standard Error

Used only for diagnostic messages.

4.55.6.3 Output Files

The output files shall be text files whose formats are dependent on the editing commands given.

4.55.7 Extended Description

The *script* shall consist of editing commands, one per line, of the following form:

[*address* [,*address*]]*command* [*arguments*]

Zero or more `<blank>`s shall be accepted before the first address and before *command.*

In default operation, `sed` cyclically shall copy a line of input, less its terminating `<newline>`, into a *pattern space* (unless there is something left after a `D` command), apply in sequence all commands whose addresses select that pattern space, and at the end of the script copy the pattern space to standard output (except when `−n` is specified) and delete the pattern space. Whenever the pattern space is written to standard output or a named file, `sed` shall immediately follow it with a `<newline>`.

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* shall each be able to hold at least 8192 bytes.

4.55.7.1 `sed` Addresses

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 4.55.7.2, preceded and followed by a delimiter, usually a slash).

A command line with no addresses shall select every pattern space.

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

A command line with two addresses shall select the inclusive range from the first pattern space that matches the first address through the next pattern space that matches the second. (If the second address is a number less than or equal to the line number first selected, only one line shall be selected.) Starting at the first line following the selected range, `sed` shall look again for the first address. Thereafter the process shall be repeated.

Editing commands can be applied only to nonselected pattern spaces by use of the negation command ! (see 4.55.7.3).

4.55.7.2 `sed` Regular Expressions

The `sed` utility shall support the basic regular expressions described in 2.8.3, with the following additions:

(1) In a context address, the construction *cREc*, where *c* is any character other than `<backslash>` or `<newline>`, shall be identical to /*RE*/ . If the character designated by *c* appears following a backslash, then it shall be considered to be that literal character, which shall 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`.

(2) The escape sequence `\n` shall match a `<newline>` embedded in the pattern space. A literal `<newline>` character shall not be used in the regular expression of a context address or in the substitute command.

4.55.7.3 `sed` Editing Commands

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* shall consist of one or more lines. Each embedded `<newline>` in the text shall be preceded by a backslash. Other backslashes in text shall be removed and the following character shall be treated literally.

The `r` and `w` commands take an optional *rfile* (or *wfile*) parameter, separated from the command letter by one or more `<blank>`s; implementations may allow zero separation as an extension.

The argument *rfile* or the argument *wfile* shall terminate the command line. Each *wfile* shall be created before processing begins. Implementations shall support at least nine *wfile* arguments in the script; the actual number (≥9) that shall be supported by the implementation is unspecified. The use of the *wfile* parameter shall cause that file to be initially created, if it does not exist, or shall replace the contents of an existing file.

The `b`, `r`, `s`, `t`, `w`, `y`, `!`, and `:` commands shall accept additional arguments. The following synopses indicate which arguments shall be separated from the commands by a single `<space>`.

Two of the commands take a *command-list* , which is a list of `sed` commands separated by `<newline>`s, as follows:

{ *command*
command
...
}

The `{` can be preceded with `<blank>`s and can be followed with white space. The *commands* can be preceded by white space. The terminating `}` shall be preceded by a `<newline>` and then zero or more `<blank>`s.

[*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. 1

[*2addr*]`b` [*label*]
Branch to the `:` command bearing the *label* . If *label* is not specified, branch to the end of the script. The implementation

shall support *labels* recognized as unique up to at least 8 characters; the actual length (≥8) that shall be 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 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 through the first `<newline>` 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>` 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>` 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 Table 2-15 (see 2.12) shall be written as the corresponding escape sequence. Nonprintable characters not in Table 2-15 shall be written as one three-digit octal number (with a preceding `<backslash>`) 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 nonprintable characters is implementation defined.

Long lines shall be folded, with the point of folding indicated by writing `<backslash><newline>`; the length at which folding occurs is unspecified, but should be appropriate for the output device. The end of each line shall 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>` 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>`, 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 shall be 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* shall 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, shall be replaced by the text matched by the corresponding backreference expression (see 2.8.3.3).

A line can be split by substituting a `<newline>` character into it. The application shall escape the `<newline>` in the *replacement* by preceding it by backslash. A substitution shall be considered to have been performed even if the replacement string is identical to the string that it replaces.

The value of *flags* shall 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 nonoverlapping 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 1
instead of slash to delimit the strings. Within *string1* and 1
string2, the delimiter itself can be used as a literal character if it 1
is preceded by a backslash. 1

[*2addr*]`!`*command*
[*2addr*]`!{`*command-list*
`}`
: Apply the *command* or *command-list* only to the lines that are not selected by the address(es).

[*0addr*]`:`*label*
: This command shall do 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*> 1

[*0addr*]
: An empty command shall be ignored.

[*0addr*]`#`
: The # and the remainder of the line shall be ignored (treated as a comment), with the single exception that if the first two characters in the file are `#n`, the default output shall be suppressed; this shall be the equivalent of specifying `−n` on the command line.

4.55.8 Exit Status

The `sed` utility shall exit with one of the following values:

0
: Successful completion.

>0
: An error occurred.

4.55.9 Consequences of Errors

Default.

4.55.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

See the rationale for `cat` (4.4.10) for an example `sed` script.

This standard requires implementations to support at least nine distinct *wfiles,* matching historical practice on many implementations. Implementations are encouraged to support more, but portable applications should not exceed this limit.

Note that regular expressions match entire strings, not just individual lines, but `<newline>` is matched by `\n` in a `sed` RE; `<newline>` is not allowed in an RE. Also note that `\n` cannot be used to match a `<newline>` at the end of an input line; `<newline>`s appear in the pattern space as a result of the `N` editing command.

The exit status codes specified here are different from those in System V. System V returns 2 for garbled `sed` commands, but returns zero with its usage message or if the input file could not be opened. The working group considered this to be a bug.

History of Decisions Made

The manner in which the `l` command writes nonprintable characters was
changed to avoid the historical backspace-overstrike method and added other 1
requirements to achieve unambiguous output. See the rationale for `ed` (4.20.10) 1
for details of the format chosen, which is the same as that chosen for `sed`. 1

The standard requires implementations to provide pattern and hold spaces of at least 8192 bytes, larger than the 4000-byte spaces used by some historical implementations, but less than the 20K byte limit used in an earlier draft. Implementations are encouraged to dynamically allocate larger pattern and hold spaces as needed.

The requirements for acceptance of `<blank>`s and `<space>`s in command lines
has been made more explicit than in earlier drafts to clearly describe existing
practice and remove confusion about the phrase ‘‘protect initial blanks [sic] and
tabs from the stripping that is done on every script line’’ that appears in much of
the historical documentation of the `sed` utility description of text. (Not all imple- 1
mentations are known to have stripped `<blank>`s from text lines, although they 1
all have allowed leading `<blank>`s preceding the address on a command line.) 1

The treatment of # comments differs from the *SVID*, which only allows a comment as the first line of the script, but matches BSD-derived implementations. The comment character is treated as a command and it has the same properties in terms of being accepted with leading `<blank>`s; the BSD implementation has historically supported this.

Earlier drafts of POSIX.2 required that a *script_file* have at least one noncomment line. Some historical implementations have behaved in unexpected ways if this were not the case. The working group felt that this was incorrect behavior, and that application developers should not have to work around this feature. A correct implementation of POSIX.2 shall permit *script_file* s that consist only of comment lines.

Earlier drafts indicated that if `−e` and `−f` options were intermixed, all `−e` options were processed before any `−f` options. This has been changed to process them in the order presented because it matches existing practice and is more intuitive.

The treatment of the `p` flag to the `s` command differs between System V and BSD-based systems (actually, between Version 7 and 32V) when the default output is suppressed. In the two examples:

```
echo a | sed    's/a/A/p'
echo a | sed -n 's/a/A/p'
```

POSIX.2, BSD, System V documentation, and the *SVID* indicate that the first example should write two lines with `A`, whereas the second should write one. Some System V systems write the `A` only once in both examples, because the `p` flag is ignored if the `−n` option is not specified.

This is a case of a diametrical difference between systems that could not be reconciled through the compromise of declaring the behavior to be unspecified. The *SVID* /BSD/32V behavior was adopted for POSIX.2 because:

— No known documentation for any historic system describes the interaction between the `p` flag and the `−n` option.

— The selected behavior is more correct as there is no technical justification for any interaction between the `p` flag and the `−n` option. A relationship between `−n` and the `p` flag might imply that they are only used together (when `p` should be a no-op), but this ignores valid scripts that interrupt the cyclical nature of the processing through the use of the `D`, `d`, `q`, or branching commands. Such scripts rely on the `p` suffix to write the pattern space because they do not make use of the default output at the ‘‘bottom’’ of the script.

— Because the `−n` option makes the `p` flag a no-op, any interaction would only be useful if `sed` scripts were written to run both with and without the `−n` option. This is believed to be unlikely. It is even more unlikely that programmers have coded the `p` flag expecting it to be a no-op. Because the interaction was not documented, the likelihood of a programmer discovering the interaction and depending on it is further decreased.

— Finally, scripts that break under the specified behavior will produce too much output instead of too little, which is easier to diagnose and correct.

The form of the substitute command that uses the *n* suffix was limited to the first 512 matches in a previous draft. This limit has been removed because there is no reason an editor processing lines of {LINE_MAX} length should have this restriction. The command `s/a/A/2047` should be able to substitute the 2047th occurrence of `a` on a line.

4.56 `sh` — Shell, the standard command language interpreter

4.56.1 Synopsis

`sh [−aCefinuvx] [` *command_file* `[` *argument* `... ] ]` 1

`sh −c [−aCefinuvx]` *command_string* `[` *command_name* `[` *argument* `... ] ]` 1

`sh −s [−aCefinuvx] [` *argument* `... ]` 1

4.56.2 Description

The `sh` utility is a command language interpreter that shall execute commands read from a command-line string, the standard input, or a specified file. The commands to be executed shall be expressed in the language described in Section 3.

4.56.3 Options

The `sh` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The `−a`, `−C`, `−e`, `−f`, `−n`, `−u`, `−v`, and `−x` options are described as part of the `set` utility in 3.14.11. The following additional options shall be supported by the implementation:

`−c` Read commands from the *command_string* operand. Set the value of special parameter `0` (see 3.5.2) from the value of the *command_name* operand and the positional parameters (`$1`, `$2`, etc.) in sequence from the remaining *argument* operands. No commands shall 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 shall be 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* . (See 3.1.4.) The behavior of an interactive shell is not fully specified by this standard.

NOTE: The preceding sentence is expected to change following the eventual approval of the UPE supplement.

Implementations may accept the option letters with a leading plus sign (`+`) instead of a leading hyphen (meaning the reverse case of the option as described in this standard). A conforming application shall protect its first operand, if it starts with a plus sign, by preceding it with the `--` argument that denotes ''end of options.''

4.56.4 Operands

The following operands shall be supported by the implementation:

`-` A single hyphen shall be 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`, etc.) shall 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 shall attempt to read that file; the file need not be executable. If the pathname does not contain a slash character:

— The implementation shall 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 3.9.1.1.

Special parameter `0` (see 3.5.2) shall be set to the value of *command_file* . If `sh` is called using a synopsis form that omits *command_file* , special parameter `0` shall be set to the value of the first argument passed to `sh` from its parent (e.g., *argv* [0] in the C binding), 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` shall be set to the value of the first argument passed to `sh` from its parent (e.g., *argv* [0] in the C binding), which is normally a pathname used to execute the `sh` utility.

command_string

A string that shall be interpreted by the shell as one or more commands, as if the string were the argument to the function in 7.1.1 [such as the *system* () function in the C binding]. If the *command_string* operand is an empty string, `sh` shall exit with a zero exit status. 1

4.56.5 External Influences

4.56.5.1 Standard Input

The standard input shall be used only if:

(1) The `-s` option is specified, or;

(2) The `-c` option is not specified and no operands are specified, or;

(3) 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 shall ensure that the standard input file pointer points
directly after the command it has read when the command begins execution. It 1
shall not read ahead in such a manner that any characters intended to be read by 1
the invoked command are consumed by the shell (whether interpreted by the shell 1
or not) or that characters that are not read by the invoked command are not seen 1
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 nonblocking 1
reads, then `sh` shall enable blocking reads on standard input. This shall remain 1
in effect when the command completes. 1

4.56.5.2 Input Files

The input file shall be a text file, except that line lengths shall be unlimited. If 1
the input file is empty or consists solely of blank lines and/or comments, `sh` shall 1
exit with a zero exit status. 1

4.56.5.3 Environment Variables

The following environment variables shall affect the execution of `sh`:

HOME This variable shall be interpreted as the pathname of the user's home directory. The contents of **HOME** are used in Tilde Expansion as described in 3.6.1.

IFS *Input field separators* : a string treated as a list of characters that shall be used for field splitting and to split lines into words with the `read` command. See 3.6.5. If **IFS** is not set, the shell shall behave 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 This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the behavior of range expressions, equivalence classes, and multicharacter collating elements within pattern matching.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files), which characters are defined as letters (character class `alpha`), and the behavior of character classes within pattern matching.

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

PATH This variable shall represent a string formatted as described in 2.6, used to effect command interpretation. See 3.9.1.1.

4.56.5.4 Asynchronous Events

Default.

4.56.6 External Effects

4.56.6.1 Standard Output

See Standard Error.

4.56.6.2 Standard Error

Except as otherwise stated (by the descriptions of any invoked utilities or in interactive mode), standard error is used only for diagnostic messages.

4.56.6.3 Output Files

None.

4.56.7 Extended Description

See Section 3.

4.56.8 Exit Status

The `sh` utility shall exit with one of the following values:

0	The script to be executed consisted solely of zero or more blank lines and/or comments.
1–125	A noninteractive shell detected a syntax, redirection, or variable assignment error.
127	A specified *command_file* could not be found by a noninteractive shell.

Otherwise, the shell shall return the exit status of the last command it invoked or attempted to invoke (see also the `exit` utility in 3.14.7).

4.56.9 Consequences of Errors

See 3.8.1.

4.56.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

```
sh -c "cat myfile"
```

```
sh my_shell_cmds
```

The `sh` utility and the `set` special built-in utility share a common set of options. Unlike `set`, however, the POSIX.2 `sh` does not specify the use of `+` as an option flag, because it is not particularly useful (the `+` variety generally invokes the default behavior) and because *getopt* () does not support it. However, since many historical implementations do support the plus, applications will have to guard against the relatively obscure case of a first operand with a leading plus sign.

There is a large number of environment variables used by historical implementations of `sh` that will not be introduced by POSIX.2 until the UPE is completed.

The KornShell ignores the contents of **IFS** upon entry to the script. A conforming application cannot rely on importing **IFS**. One justification for this, beyond security considerations, is to assist possible future shell compilers. Allowing **IFS** to be imported from the environment will prevent many optimizations that might otherwise be performed via dataflow analysis of the script itself.

The 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 noninteractive shells, respectively. Although both accept terminal input, the results of error conditions will be different, as described in 3.8.1; in the second example a redirection error encountered by a special built-in utility will abort the shell.

The text in Standard Input about nonblocking reads concerns an instance of `sh` 1
that has been invoked, probably by a C-language program, with standard input 1
that has been opened using the O_NONBLOCK flag; see POSIX.1 {8} *open*(). If the 1
shell did not reset this flag, it would immediately terminate because no input data 1
would be available yet and that would be considered the same as end-of-file. 1

History of Decisions Made

See the Rationale for Section 3 concerning the lack of interactive features in `sh`. These features, including optional job control, are scheduled to be added in the User Portability Extension.

The **PS1** and **PS2** variables are not specified because this standard, without UPE, does not describe an interactive shell.

The options associated with a *restricted shell* (command name `rsh` and the `-r` option) were excluded because the developers of the standard felt that the implied level of security was not achievable and they did not want to raise false expectations.

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, POSIX.2 allows the single hyphen to mark the end of the options, in addition to the use of the regular `--` argument, because it was felt that the older practice was so pervasive. An alternative approach is taken by the KornShell, where real and effective user/group IDs must match for an interactive shell; this behavior is specifically allowed by POSIX.2. (Note: there are other problems with set-user-ID scripts that the two approaches described here do not deal with.)

4.57 `sleep` — Suspend execution for an interval

4.57.1 Synopsis

`sleep` *time*

4.57.2 Description

The `sleep` utility shall suspend execution for at least the integral number of seconds specified by the *time* operand.

4.57.3 Options

None.

4.57.4 Operands

The following operands shall be supported by the implementation:

time A nonnegative decimal integer specifying the number of seconds for which to suspend execution.

4.57.5 External Influences

4.57.5.1 Standard Input

None.

4.57.5.2 Input Files

None.

4.57.5.3 Environment Variables

The following environment variables shall affect the execution of `sleep`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.57.5.4 Asynchronous Events

If the `sleep` utility receives a SIGALRM signal, one of the following actions shall be taken:

(1) Terminate normally with a zero exit status

(2) Effectively ignore the signal

(3) Provide the default behavior for signals described in 2.11.5.4. This could include terminating with a nonzero exit status.

The `sleep` utility shall take the standard action for all other signals; see 2.11.5.4.

4.57.6 External Effects

4.57.6.1 Standard Output

None.

4.57.6.2 Standard Error

Used only for diagnostic messages.

4.57.6.3 Output Files

None.

4.57.7 Extended Description

None.

4.57.8 Exit Status

The `sleep` utility shall exit with one of the following values:

0 The execution was successfully suspended for at least *time* seconds, or a SIGALRM signal was received (see 4.57.5.4).

>0 An error occurred.

4.57.9 Consequences of Errors

Default.

4.57.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The exit status is allowed to be zero when `sleep` is interrupted by the SIGALRM signal, because most implementations of this utility rely on the arrival of that signal to notify them that the requested finishing time has been successfully attained. Such implementations thus do not distinguish this situation from the successful completion case. Other implementations are allowed to catch the signal and go back to sleep until the requested time expires or provide the normal signal termination procedures.

History of Decisions Made

As with all other utilities that take integral operands and do not specify subranges of allowed values, `sleep` is required by this standard to deal with *time* requests of up to 2 147 483 647 seconds. This may mean that some implementations will have to make multiple calls to the underlying operating system's delay mechanism if its argument range is less than this.

4.58 `sort` — Sort, merge, or sequence check text files

4.58.1 Synopsis

`sort [−m] [−o` *output* `] [−bdfinru] [−t` *char* `] [−k` *keydef* `]` . . . [*file* . . .]

`sort −c [−bdfinru] [−t` *char* `] [−k` *keydef* `]` . . . [*file*]

Obsolescent Versions :

`sort [−mu] [−o` *output* `] [−bdfinr] [−t` *char* `]` [+*pos1* [−*pos2*]] . . . [*file* . . .]

`sort −c [−u] [−bdfinr] [−t` *char* `]` [+*pos1* [−*pos2*]] . . . [*file*]

4.58.2 Description

The `sort` utility shall perform one of the following functions:

(1) Sort lines of all the named files together and write the result to the specified output.

(2) Merge lines of all the named (presorted) files together and write the result to the specified output.

(3) Check that a single input file is correctly presorted.

Comparisons shall be based on one or more sort keys extracted from each line of input (or the entire line if no sort keys are specified), and shall be performed using the collating sequence of the current locale.

4.58.3 Options

The `sort` utility shall conform to the utility argument syntax guidelines described in 2.10.2, except that the notation *+pos1 −pos2* uses a nonstandard prefix and multidigit option names in the obsolescent versions, the `−o` *output* option shall be recognized 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 shall be supported by the implementation:

`−c` Check that the single input file is ordered as specified by the arguments and the collating sequence of the current locale. No output shall be produced; only the exit code shall be affected.

`−m` Merge only; the input files shall be 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.

The following options shall override the default ordering rules. When ordering options appear independent of any key field specifications, the requested field ordering rules shall be applied globally to all sort keys. When attached to a specific key (see `−k`), the specified ordering options shall override all global ordering options for that key. In the obsolescent forms, if one or more of these options follows a *+pos1* option, it shall affect only the key field specified by that preceding option.

`−d` Specify that only `<blank>`s and alphanumeric characters, according to the current setting of LC_CTYPE , shall be significant in comparisons. The behavior is undefined for a sort key to which `−i` or `−n` also applies.

`−f` Consider all lowercase characters that have uppercase equivalents, according to the current setting of LC_CTYPE , to be the uppercase equivalent for the purposes of comparison.

`−i` Ignore all characters that are nonprintable, according to the current setting of LC_CTYPE.

−n Restrict the sort key to an initial numeric string, consisting of optional <blank>s, optional minus sign, and zero or more digits with an optional radix character and thousands separators (as defined in the current locale), which shall be sorted by arithmetic value. An empty digit string shall be treated as zero. Leading zeros and signs on zeros shall not affect ordering.

−r Reverse the sense of comparisons.

The treatment of field separators can be altered using the options:

−b Ignore leading <blank>s when determining the starting and ending positions of a restricted sort key. If the −b option is specified before the first −k option, it shall be 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* shall not be considered to be part of a field (although it can be included in a sort key). Each occurrence of *char* shall be significant (for example, <*char*><*char*> shall delimit an empty field). If −t is not specified, <blank> characters shall be used as default field separators; each maximal nonempty sequence of <blank> characters that follows a non-<blank> character shall be 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 4.58.7), and *type* is a modifier from the list of characters b, d, f, i, n, r. The b modifier shall behave like the −b option, but applies only to the *field_start* or *field_end* to which it is attached. The other modifiers shall behave like the corresponding options, but shall apply only to the key field to which they are attached; they shall have this effect if specified with *field_start,* *field_end,* or both. Modifiers attached to a *field_start* or *field_end* shall override any specifications made by the options. Implementations shall support at least nine occurrences of the −k option, which shall be significant in command line order. If no −k option is specified, a default sort key of the entire line shall be used.

When there are multiple key fields, later keys shall be compared only after all earlier keys compare equal. Except when the −u option is specified, lines that otherwise compare equal shall be 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.

+pos1 (Obsolescent.) Specify the start position of a key field. See 4.58.7.

−pos2 (Obsolescent.) Specify the end position of a key field. See 4.58.7.

4.58.4 Operands

The following operand shall be supported by the implementation:

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 shall be used.

4.58.5 External Influences

4.58.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified, or if a *file* operand is −. See Input Files.

4.58.5.2 Input Files

The input files shall be text files, except that the `sort` utility shall add a `<newline>` to the end of a file ending with an incomplete last line.

4.58.5.3 Environment Variables

The following environment variables shall affect the execution of `sort`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for ordering rules.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and the behavior of character classification for the `−b`, `−d`, `−f`, `−i`, and `−n` options.

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

LC_NUMERIC This variable shall determine the locale for the definition of the radix character and thousands separator for the `-n` option.

4.58.5.4 Asynchronous Events

Default.

4.58.6 External Effects

4.58.6.1 Standard Output

Unless the `-o` or `-c` options are in effect, the standard output shall contain the sorted input.

4.58.6.2 Standard Error

Used only for diagnostic messages. A warning message about correcting an 2
incomplete last line of an input file may be generated, but need not affect the final 2
exit status. 2

4.58.6.3 Output Files

If the `-o` option is in effect, the sorted input shall be placed in the file *output* .

4.58.7 Extended Description

The notation

`-k` *field_start* [*type*][`,`*field_end* [*type*]]

shall define 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 shall be empty. A missing *field_end* shall mean the last character of the line.

A field comprises a maximal sequence of nonseparating characters and, in the 1
absence of option `-t`, any preceding field separator. 1

The *field_start* portion of the *keydef* option argument shall have the form:

field_number [`.`*first_character*]

Fields and characters within fields shall be numbered starting with 1. The *field_number* and *first_character* pieces, interpreted as positive decimal integers, shall specify the first character to be used as part of a sort key. If `.`*first_character* is omitted, it shall refer to the first character of the field.

The *field_end* portion of the *keydef* option argument shall have the form:

field_number [.*last_character*]

The *field_number* shall be as described above for *field_start*. The *last_character* piece, interpreted as a nonnegative decimal integer, shall specify the last character to be used as part of the sort key. If *last_character* evaluates to zero or .*last_character* is omitted, it shall refer 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 shall be counted from the first non-<blank> in the field. (This shall apply separately to *first_character* and *last_character* .)

The obsolescent [+*pos1* [−*pos2*]] options provide functionality equivalent to the −k *keydef* option. For comparison, the full formats of these options shall be:

```
+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*) shall be numbered from zero instead of one. The −*pos2* option shall specify the first character after the sort field instead of the last character in the sort field. (Therefore, *field0_number* and *first0_character* shall be interpreted as nonnegative, instead of positive, decimal integers and there is no need for a specification of a *last_character* -like form.) The optional type modifiers shall be the same in both forms. If .*first0_character* is omitted or *first0_character* evaluates to zero, it shall refer to the first character of the field.

Thus, a the fully specified +*pos1* −*pos2* form:

```
+w.x -y.z
```

shall be equivalent to:

```
-k w+1.x+1,y.0     (if z == 0)
-k w+1.x+1,y+1.z   (if z > 0)
```

As with the nonobsolescent forms, implementations shall support at least nine occurrences of the +*pos1* option, which shall be significant in command line order.

4.58.8 Exit Status

The sort utility shall exit with one of the following values:

- 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. This exit status shall not be returned if the −c option is not used.
- >1 An error occurred.

4.58.9 Consequences of Errors

Default.

4.58.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

In the following examples, nonobsolescent and obsolescent ways of specifying sort keys are given as an aid to understanding the relationship between the two forms.

Either of the following commands sorts the contents of `infile` with the second field as the sort key:

```
sort -k 2,2 infile
sort +1 -2 infile
```

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        1
sort -r -o outfile +1.1 -1.2 infile1 infile2
```

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
sort +1.1b -1.2b  infile1 infile2
```

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

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
sort -um +2.0 -3.0 infile
```

Examples in some historical documentation state that options `–um` with one input 1
file keep the first in each set of lines with equal keys. This behavior was deemed 2
to be an implementation artifact and was not made standard. 1

The default value for `–t`, `<blank>`, has different properties than, for example, `-t "<space>"`. If a line contains:

```
<space><space>foo
```

the following treatment would occur with default separation versus specifically selecting a `<space>`:

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 <tab> precedes <space> 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 nonnumeric field separator.

History of Decisions Made

The −z option was removed; it is not standard practice on most systems, and is inconsistent with using sort to individually sort several files and then merging them together. The previous language appeared to require implementations to determine the proper buffer length during the sort phase of operation, but not during the merge.

The −y option was removed because of nonportability. The −M option, present in System V, was removed because of nonportability in international usage.

An undocumented −T option exists in some implementations. It is used to specify a directory for intermediate files. Implementations are encouraged to support the use of the **TMPDIR** environment variable instead of adding an option to support this functionality.

The −k option was added to satisfy two complaints. First, the zero-based counting used by sort is not consistent with other utility conventions. Second, it did not meet syntax guideline requirements. The one-based counting in this standard was developed from the input provided by several ballot comments, ballot objections, and discussions with users.

The wording in Draft 10 also 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 standard matches historical practice, not historical documentation. In the nonobsolescent versions, the results are unspecified if these options are specified after a `−k` option. This will allow implementations to make the options independent of each other when the obsolescent forms are finally dropped (if that ever happens).

Historical documentation indicates that ''setting `−n` implies `−b`.'' The description of `−n` already states that optional leading `<blank>`s are tolerated in doing the comparison. If `−b` is enabled, rather than implied, by `−n`, this has unusual side effects. When a character offset is used into a column of numbers (e.g., to sort mod 100), that offset will be measured relative to the most significant digit, not to the column. Based upon a recommendation of the author of the original `sort` utility, the `−b` implication has been omitted from POSIX.2 and an application wishing to achieve the previously mentioned side effects will have to manually code the `−b` flag.

4.59 `stty` — Set the options for a terminal

4.59.1 Synopsis

`stty [ −a | −g ]`

`stty` *operands*

4.59.2 Description

The `stty` utility shall set or report on terminal I/O characteristics for the device that is its standard input. Without options or operands specified, it shall report the settings of certain characteristics, usually those that differ from implementation-defined defaults. Otherwise, it shall modify the terminal state according to the specified operands. Detailed information about the modes listed in the first five groups below are described in POSIX.1 {8} Section 7. Operands in the Combination Modes group (see 4.59.4.6) shall be 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 POSIX.1 {8} *termios* interface. On systems where none of these lines are available, and on lines not currently configured to support the POSIX.1 {8} termios interface, some of the operands need not affect terminal characteristics.

4.59.3 Options

The `stty` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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 shall not contain any characters that would require quoting to avoid word expansion by the shell; see 3.6.

4.59.4 Operands

The following operands shall be supported by the implementation to set the terminal characteristics:

4.59.4.1 Control Modes

`parenb (−parenb)` Enable (disable) parity generation and detection. This shall have the effect of setting (not setting) PARENB in the *termios c_cflag* field, as defined in POSIX.1 {8}.

`parodd (−parodd)` Select odd (even) parity. This shall have the effect of setting (not setting) PARODD in the *termios c_cflag* field, as defined in POSIX.1 {8}.

`cs5 cs6 cs7 cs8` Select character size, if possible. This shall have the effect of setting CS5, CS6, CS7, and CS8, respectively, in the *termios c_cflag* field, as defined in POSIX.1 {8}.

number Set terminal baud rate to the number given, if possible. If the baud rate is set to zero, the modem control lines shall no longer be asserted. This shall have the effect of setting the input and output *termios* baud rate values as defined in POSIX.1 {8}.

`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 shall be specified by the value of the output baud rate. This shall have the effect of setting the input *termios* baud rate values as defined in POSIX.1 {8}.

`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 shall no longer be asserted. This shall have the effect of setting the output *termios* baud rate values as defined in POSIX.1 {8}.

`hupcl (−hupcl)` Stop asserting modem control lines (do not stop asserting modem control lines) on last close. This shall have the effect of setting (not setting) HUPCL in the *termios c_cflag* field, as defined in POSIX.1 {8}.

`hup (−hup)` Same as `hupcl` (`−hupcl`).

`cstopb (−cstopb)` Use two (one) stop bits per character. This shall have the effect of setting (not setting) CSTOPB in the *termios c_cflag* field, as defined in POSIX.1 {8}.

`cread (−cread)` Enable (disable) the receiver. This shall have the effect of setting (not setting) CREAD in the *termios c_cflag* field, as defined in POSIX.1 {8}.

`clocal (−clocal)` Assume a line without (with) modem control. This shall have the effect of setting (not setting) CLOCAL in the *termios c_cflag* field, as defined in POSIX.1 {8}.

It is unspecified whether `stty` shall report an error if an attempt to set a Control Mode fails.

4.59.4.2 Input Modes

`ignbrk (−ignbrk)` Ignore (do not ignore) break on input. This shall have the effect of setting (not setting) IGNBRK in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`brkint (−brkint)` Signal (do not signal) INTR on break. This shall have the effect of setting (not setting) BRKINT in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`ignpar (−ignpar)` Ignore (do not ignore) bytes with parity errors. This shall have the effect of setting (not setting) IGNPAR in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`parmrk (−parmrk)` Mark (do not mark) parity errors. This shall have the effect of setting (not setting) PARMRK in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`inpck (−inpck)` Enable (disable) input parity checking. This shall have the effect of setting (not setting) INPCK in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`istrip (−istrip)` Strip (do not strip) input characters to seven bits. This shall have the effect of setting (not setting) ISTRIP in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`inlcr (−inlcr)` Map (do not map) NL to CR on input. This shall have the effect of setting (not setting) INLCR in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`igncr (−igncr)` Ignore (do not ignore) CR on input. This shall have the effect of setting (not setting) IGNCR in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`icrnl (−icrnl)` Map (do not map) CR to NL on input. This shall have the effect of setting (not setting) ICRNL in the *termios* *c_iflag* field, as defined in POSIX.1 {8}.

`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 shall have the effect of setting (not setting) IXON in the *termios c_iflag* field, as defined in POSIX.1 {8}.

`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 shall have the effect of setting (not setting) IXOFF in the *termios c_iflag* field, as defined in POSIX.1 {8}.

4.59.4.3 Output Modes

`opost (−opost)` Post-process output (do not post-process output; ignore all other output modes). This shall have the effect of setting (not setting) OPOST in the *termios c_oflag* field, as defined in POSIX.1 {8}.

4.59.4.4 Local Modes

`isig (−isig)` Enable (disable) the checking of characters against the special control characters INTR, QUIT, and SUSP. This shall have the effect of setting (not setting) ISIG in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`icanon (−icanon)` Enable (disable) canonical input (ERASE and KILL processing). This shall have the effect of setting (not setting) ICANON in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`iexten (−iexten)` Enable (disable) any implementation-defined special control characters not currently controlled by `icanon`, `isig`, `ixon`, or `ixoff`. This shall have the effect of setting (not setting) IEXTEN in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`echo (−echo)` Echo back (do not echo back) every character typed. This shall have the effect of setting (not setting) ECHO in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`echoe (−echoe)` The ERASE character shall (shall not) visually erase the last character in the current line from the display, if possible. This shall have the effect of setting (not setting) ECHOE in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`echok (−echok)` Echo (do not echo) NL after KILL character. This shall have the effect of setting (not setting) ECHOK in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`echonl (−echonl)` Echo (do not echo) NL, even if `echo` is disabled. This shall have the effect of setting (not setting) ECHONL in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`noflsh (−noflsh)` Disable (enable) flush after INTR, QUIT, SUSP. This shall have the effect of setting (not setting) NOFLSH in the *termios c_lflag* field, as defined in POSIX.1 {8}.

`tostop (−tostop)` Send SIGTTOU for background output. This shall 2
have the effect of setting (not setting) TOSTOP in the 2
termios c_lflag field, as defined in POSIX.1 {8}. 2

NOTE: Setting TOSTOP has no effect on systems not supporting 2
the POSIX.1 {8} job control option. 2

4.59.4.5 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 Table 4-9, the corresponding POSIX.1 {8} control character from the second column shall be recognized. This shall have the effect of setting the corresponding element of the *termios c_cc* array (see POSIX.1 {8} 7.1.2).

Table 4-9 − `stty` Control Character Names

control-character	POSIX.1 {8} 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

If *string* is a single character, the control character shall be set to that character. If *string* is the two-character sequence `"^−"` or the string `"undef"`, the control character shall be set to {_POSIX_VDISABLE}, if it is in effect for the device; if {_POSIX_VDISABLE} is not in effect for the device, it shall 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 Table 4-10, the control character shall be set to the corresponding character value in the Value column of the table.

Table 4-10 – `stty` Circumflex Control Characters

^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>		

`min` *number*

`time` *number* — Set the value of `min` or `time` to *number*. MIN and TIME are used in noncanonical mode input processing (`−icanon`).

4.59.4.6 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`.

`nl (−nl)` — Enable (disable) `icrnl`. In addition, `−nl` unsets `inlcr` and `igncr`.

`ek` — Reset ERASE and KILL characters back to system defaults.

`sane` — Reset all modes to some reasonable, unspecified, values.

4.59.5 External Influences

4.59.5.1 Standard Input

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.

4.59.5.2 Input Files

None.

4.59.5.3 Environment Variables

The following environment variables shall affect the execution of `stty`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and which characters are in the class `print`.

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

4.59.5.4 Asynchronous Events

Default.

4.59.6 External Effects

4.59.6.1 Standard Output

If operands are specified, no output shall be produced.

If the `-g` option is specified, `stty` shall 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 4.59.4 shall be written to standard output. Unless otherwise specified, this information shall be 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 shall be 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 shall be written as follows:

`"speed %d baud;"`, <*speed*>

Otherwise, speeds shall 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 shall 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 nonprintable, or the string `<undef>` if the character is disabled.

4.59.6.2 Standard Error

Used only for diagnostic messages.

4.59.6.3 Output Files

None.

4.59.7 Extended Description

None.

4.59.8 Exit Status

The `stty` utility shall exit with one of the following values:

0 The terminal options were read or set successfully.

>0 An error occurred.

4.59.9 Consequences of Errors

Default.

4.59.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Since POSIX.1 {8} doesn't specify any output modes, they are not specified in this standard either. Implementations are expected to provide `stty` operands corresponding to all of the output modes they support.

In many ways outside the scope of POSIX.2, `stty` is primarily used to tailor the user interface of the terminal, such as selecting the preferred ERASE and KILL characters. As an application programming utility, `stty` can be used within shell scripts to alter the terminal settings for the duration of the script. 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.

History of Decisions Made

The original `stty` manual page was taken directly from System V and reflected the System V terminal driver *termio*. It has been modified to correspond to the POSIX.1 {8} terminal driver *termios*.

The *termios* section states that individual disabling of control characters is an option {_POSIX_VDISABLE}. If enabled, two conventions currently exist for specifying this: System V uses `"^-"`, and BSD uses `undef`. Both are accepted by POSIX.2 `stty`. The other BSD convention of using the letter `u` was rejected because it conflicts with the actual letter `u`, which is an acceptable value for a control character.

Early drafts did not specify the mapping of *^c* to control characters because the control characters were not specified in the POSIX Locale character set description file requirements. The control character set is now specified in 2.4.1, so the traditional mapping is specified. Note that although the mapping corresponds to control-character key assignments on many terminals that use ISO/IEC 646 {1} (or ASCII) character encodings, the mapping specified here is to the control characters, not their keyboard encodings.

The combination options `raw` and `cooked` (`−raw`) were dropped from the standard because the exact values that should be set are not well understood or commonly agreed on. In particular, *termios* has no explicit RAW bit, and the options that should be re-enabled (`−raw`) are not clear. General programming practice is to save the terminal state, change the settings for the duration of the program, and then reset the state. This is easy to do within a C program, however it is not possible for a single invocation of `stty` to restore the terminal state (`−raw`) without knowledge of the prior settings. Using the `−g` option and two calls to `stty`, a shell application could do this as described above. However, it is impossible to implement this as a single option. Also, it is not clear that changing word size and parity is appropriate. For example, requiring that `cooked` set `cs7` and `parenb` would be disastrous for users working with 8−bit international character sets. In general, these options are too ill-defined to be of any use.

Since *termios* supports separate speeds for input and output, two new options were added to specify each distinctly.

The `ixany` input mode was removed from Draft 8 on the basis that it could not be implemented on a POSIX.1 {8} system without extensions.

Some historical implementations use standard input to get and set terminal characteristics; others use standard output. Since input from a login TTY is usually restricted to the owner while output to a TTY is frequently open to the world, using standard input provides fewer chances of accidentally (or mischievously) altering the terminal settings of other users. Using standard input also allows `stty −a` and `stty −g` output to be redirected for later use. Therefore, usage of standard input is required by this standard.

The `tostop` option was omitted from early drafts through an oversight. It is the only option that requires job control to be effective, and thus could have gone into the UPE as a modification to `stty`, but since all other terminal control features are in the base standard, `tostop` was included as well.

4.60 **`tail`** — Copy the last part of a file

4.60.1 Synopsis

`tail` `[−f]` `[` `−c` *number* **|** `−n` *number* `]` `[`*file*`]`

Obsolescent versions:

`tail` `−[`*number*`][c|l][f]` `[`*file*`]`

`tail` `+[`*number*`][c|l][f]` `[`*file*`]`

4.60.2 Description

The `tail` utility shall copy its input file to the standard output beginning at a designated place.

Copying shall begin 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* shall be counted in units of lines or bytes, according to the options `−n` and `−c` (or, in the obsolescent version, the appended option suffixes `l` or `c`).

Tails relative to the end of the file may be saved in an internal buffer, and thus may be limited in length. Implementations shall ensure that such a buffer, if any, is no smaller than {LINE_MAX}*10 bytes.

4.60.3 Options

The `tail` utility shall conform to the utility argument syntax guidelines described in standard described in 2.10.2, except that the obsolescent version accepts multicharacter options that can preceded by a plus sign.

The following options shall be supported by the implementation in the nonobsolescent version:

`−c` *number* The *number* option-argument shall 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 shall be 1; i.e., `-c +1` represents the first 1
byte of the file, `-c -1` the last. 1

`−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 shall be ignored. If the input file is not a FIFO, pipe, or regular file, it is unspecified whether or not the `−f` option shall be ignored.

`−n` *number* This option shall be equivalent to `−c` *number*, except the starting
location in the file shall be measured in lines instead of bytes. 1
The origin for counting shall be 1; i.e., `-n +1` represents the first 1
line of the file, `-n -1` the last. 1

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 shall be equivalent to `−c` ±*number*; ±*number* with the letter `l` specified as a suffix, or with neither `c` nor `l` as a suffix, shall be equivalent to `−n` ±*number*. If *number* is not specified in these forms, 10 shall be used. The letter `f` specified as a suffix shall be equivalent to specifying the `−f` option. If the `-[`*number*`]c[f]` form is used and neither *number* nor the `f` suffix is specified, it shall be interpreted as the `−c` *number* option.

In the nonobsolescent form, if neither `−c` nor `−n` is specified, `-n 10` shall be assumed.

4.60.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

4.60.5 External Influences

4.60.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.60.5.2 Input Files

If the `−c` option is specified, the input file can contain arbitrary data; otherwise, the input file shall be a text file.

4.60.5.3 Environment Variables

The following environment variables shall affect the execution of `tail`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.60.5.4 Asynchronous Events

Default.

4.60.6 External Effects

4.60.6.1 Standard Output

The designated portion of the input file shall be written to standard output.

4.60.6.2 Standard Error

Used only for diagnostic messages.

4.60.6.3 Output Files

None.

4.60.7 Extended Description

None.

4.60.8 Exit Status

The `tail` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

4.60.9 Consequences of Errors

Default.

4.60.10 Rationale. *(This subclause is not a part of P1003.2)*

Usage, Examples

The nonobsolescent version of `tail` was created to allow conformance to the Utility Syntax Guidelines. The historical `−b` option was omitted because of the general nonportability of block-sized units of text. The `−c` option historically meant ‘‘characters,’’ but this standard indicates that it means ‘‘bytes.’’ This was selected to allow reasonable implementations when multibyte characters are possible; it was not named `−b` to avoid confusion with the historical `−b`.

Note that the `−c` option should be used with caution when the input is a text file containing multibyte characters; it may produce output that does not start on a character boundary.

The origin of counting both lines and bytes is 1, matching all widespread histori- 1
cal implementations. 1

The restriction on the internal buffer is a compromise between the historical System V implementation of 4K and the BSD 32K.

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.

Although the input file to `tail` can be any type, the results need not be what would be expected on some character special device files or on file types not described by POSIX.1 {8}. Since the standard 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.

History of Decisions Made

The developers of the standard originally decided that `tail`, and its frequent companion, `head`, were useful mostly to interactive users, and not application programs. However, balloting input suggested that these utilities actually do find significant use in scripts, such as to write out portions of log files. The balloters also challenged the working group's assumption that clever use of `sed` could be an appropriate substitute for `tail`.

The `−f` option has been implemented as a loop that sleeps for one second and copies any bytes that are available. This is sufficient, but if more efficient methods of determining when new data are available are developed, implementations are encouraged to use them.

Historical documentation says that `tail` ignores the `−f` option if the input file is a pipe (pipe and FIFO on systems that support FIFOs). On BSD-based systems, this has been true; on System V-based systems, this was true when input was taken from standard input, but behaved as on other files if a FIFO was named as the *file* operand. Since the `−f` option is not useful on pipes and all historical implementations ignore `−f` if no *file* operand is specified and standard input is a pipe, POSIX.2 requires this behavior. However, since the `−f` option is useful on a FIFO, POSIX.2 also requires that if standard input is a FIFO or a FIFO is named, the `−f` option shall not be ignored. Although historical behavior does not ignore the `−f` option for other file types, this is unspecified so that implementations are allowed to ignore the `−f` option if it is known that the file cannot be extended.

An earlier draft had the synopsis line:

`tail [−c | −l] [−f] [−n` *number* `] [`*file*`]`

This was changed to the current form based on comments and objections noting that `−c` was almost never used without specifying a number and there was no need to specify `−l` if `−n` *number* was given.

4.61 `tee` — Duplicate standard input

4.61.1 Synopsis

`tee [−ai] [`*file* ...`]`

4.61.2 Description

The `tee` utility shall copy standard input to standard output, making a copy in zero or more files. The `tee` utility shall not buffer output.

The options determine if the specified files are overwritten or appended to.

4.61.3 Options

The `tee` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−a` Append the output to the files rather than overwriting them.

`−i` Ignore the SIGINT signal.

4.61.4 Operands

The following operands shall be supported by the implementation:

file A pathname of an output file. Implementations shall support processing of at least 13 *file* operands.

4.61.5 External Influences

4.61.5.1 Standard Input

The standard input can be of any type.

4.61.5.2 Input Files

None.

4.61.5.3 Environment Variables

The following environment variables shall affect the execution of `tee`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.61.5.4 Asynchronous Events

Default, except that if the `-i` option was specified, SIGINT shall be ignored.

4.61.6 External Effects

4.61.6.1 Standard Output

The standard output shall be a copy of the standard input.

4.61.6.2 Standard Error

Used only for diagnostic messages.

4.61.6.3 Output Files

If any *file* operands are specified, the standard input shall be copied to each named file.

4.61.7 Extended Description

None.

4.61.8 Exit Status

0 The standard input was successfully copied to all output files.

>0 An error occurred.

4.61.9 Consequences of Errors

If a write to any successfully opened *file* operand fails, writes to other successfully opened *file* operands and standard output shall continue, but the exit status shall be nonzero. Otherwise, the default actions specified in 2.11.9 shall apply.

4.61.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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.

History of Decisions Made

The buffering requirement means that `tee` is not allowed to use C Standard {7} fully-buffered or line-buffered writes, not that `tee` has to do one-byte reads followed by one-byte writes.

It should be noted that early versions of BSD silently ignore any invalid options, and accept a single `–` as an alternative to `–i`. They also print the message

`"tee: cannot access %s\n"`, <*pathname*>

if unable to open a file.

Historical implementations ignore write errors. This is explicitly not permitted by this standard.

Some historical implementations use O_APPEND when providing append mode; others just *lseek*() to the end of file after opening the file without O_APPEND. This standard requires functionality equivalent to using O_APPEND; see 2.9.1.4.

4.62 `test` — Evaluate expression

4.62.1 Synopsis

`test` [*expression*]

`[` [*expression*] `]`

4.62.2 Description

The `test` utility shall evaluate the *expression* and indicate the result of the 1
evaluation by its exit status. An exit status of zero indicates that the expression 1
evaluated as true and an exit status of 1 indicates that the expression evaluated 1
as false. 1

In the second form of the utility, which uses `[ ]`, rather than `test`, the square brackets shall be separate arguments.

4.62.3 Options

The `test` utility shall not recognize the `--` argument in the manner specified by utility syntax guideline 10 in 2.10.2.

Implementations shall not support any options. 1

4.62.4 Operands

All operators and elements of primaries shall be presented as separate arguments 2
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 nonzero.
`−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 shall indicate only that the write flag is on. The *file* shall 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 shall 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* .

A primary can be preceded by the ! operator to complement its test, as described 1
below. 1

The primaries with two elements of the form: 2

−*primary_operator* *primary_operand* 2

are known as *unary primaries.* The primaries with three elements in either of the 2
two forms: 2

primary_operand −*primary_operator* *primary_operand* 2

primary_operand *primary_operator* *primary_operand* 2

are known as *binary primaries.* Additional implementation-defined operators and 2
primary_operator s may be provided by implementations. They shall be of the 2
form −*operator* where the first character of *operator* is not a digit. The additional 2
implementation-defined operators ‘‘`(`’’and ‘‘`)`’’may also be provided by implemen- 2
tations. 2

The algorithm for determining the precedence of the operators and the return value that shall be generated is based on the number of arguments presented to `test`. (However, when using the `[...]` form, the right-bracket final argument shall 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`.
- Otherwise, produce unspecified results.

4 arguments:

- If `$1` is `!`, negate the three-argument test of `$2`, `$3`, and `$4`.
- Otherwise, the results are unspecified.

>4 arguments:

The results are unspecified.

4.62.5 External Influences

4.62.5.1 Standard Input

None.

4.62.5.2 Input Files

None.

4.62.5.3 Environment Variables

The following environment variables shall affect the execution of `test`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.62.5.4 Asynchronous Events

Default.

4.62.6 External Effects

4.62.6.1 Standard Output

None.

4.62.6.2 Standard Error

Used only for diagnostic messages.

4.62.6.3 Output Files

None.

4.62.7 Extended Description

None.

4.62.8 Exit Status

The `test` utility shall exit with one of the following values:

0	*expression* evaluated to true.
1	*expression* evaluated to false or *expression* was missing.
>1	An error occurred.

4.62.9 Consequences of Errors

Default.

4.62.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Editor's Note: The rationale has been rearranged quite a bit. Only new, not moved, text has been diffmarked.

Historical systems have supported more than four arguments, but there has been a fundamental disagreement between BSD and System V on certain combinations of arguments. Since no accommodation could be reached between the two versions of `test` without breaking numerous applications, the version of `test` in POSIX.2 specifies only the relatively simple tests and relies on the syntax of the shell command language for the construction of more complex expressions. Using the POSIX.2 rules produces completely reliable, portable scripts, which is not always possible using either of the historical forms. Some of the historical behavior is described here to aid conversion of scripts with complex `test` expressions.

Both BSD and System V support the combining of primaries with the following constructs:

expression1 `-a` *expression2*	True if both *expression1* and *expression2* are true.
expression1 `-o` *expression2*	True if at least one of *expression1* and *expression2* are true.
`(` *expression* `)`	True if *expression* is true.

In evaluating these more complex combined expressions, the following precedence rules are used:

- The unary primaries have higher precedence than the algebraic binary primaries.
- On BSD systems, the unary primaries have higher precedence than the string binary primaries. On System V systems, the unary primaries have lower precedence than the string binary primaries.

— The unary and binary primaries have higher precedence than the unary *string* primary.

— The ! operator has higher precedence than the `-a` operator and the `-a` operator has higher precedence than the `-o` operator.

— The `-a` and `-o` operators are left associative.

— The parentheses can be used to alter the normal precedence and associativity.

The following guidance is offered for the use of the historical expressions:

— 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 portability between implementations based on BSD and System V systems 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. Historical `test` implementations also support parentheses, but they must be escaped when using `sh`; for example:

```
test \( expr1 -a expr2 \) -o expr3
```

This command is not always portable. 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 historical systems. Unexpected results would occur if such a *string* expression were used and `$1` expanded to `!`, `(`, or a known unary primary. Better constructs were:

```
test -n "$1"
test -z "$1"
```

respectively. These suggested replacements have always worked on historical BSD-based implementations, and work on historical System V-based implementations as long as `$1` does not expand to `=` or `!=`. Using the POSIX.2 rules, any of the four forms shown will work for any possible value of `$1`.

— Historical systems were also unreliable given the common construct:

```
test "$response" = "expected string"
```

One of the following was 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 any unary primary. If `expected string` starts with `-`, `(`, `!`, or even `=`, the first form should be used instead. Using the POSIX.2 rules, any of the three comparison forms is reliable, given any input. (However, note that the strings are quoted in all cases.)

The BSD and System V versions of `−f` are not the same. The BSD definition was:

`−f` *file* True if *file* exists and is not a directory.

The *SVID* version (true if the file exists and is a regular file) was chosen for this standard because its use is consistent with the `−b`, `−c`, `−d`, and `−p` operands (*file* exists and is a specific file type).

The `−e` primary, possessing similar functionality to that provided by the C-shell, was added because it provides the only way for a shell script to find out if a file exists without trying to open the file. (Since implementations are allowed to add additional file types, a portable script cannot use:

```
test -b foo -o -c foo -o -d foo -o -f foo -o -p foo
```

to find out if `foo` is an existing file.) On historical BSD systems, the existence of a file could be determined by:

```
test -f foo -o -d foo
```

but there was no easy way to determine that an existing file was a regular file. An earlier draft used the KornShell `−a` primary (with the same meaning), but this was changed to `−e` because there were concerns about the high probability of humans confusing the `−a` primary with the `−a` binary operator.

History of Decisions Made

The `-a` and `-o` binary operators and the grouping parentheses were omitted from POSIX.2 due to a difference between existing implementations of the `test` utility in the precedence of the binary primaries `=` and `!=` compared to the unary primaries `-b`, `-c`, `-d`, `-f`, `-g`, `-n`, `-p`, `-r`, `-s`, `-t`, `-u`, `-w`, `-x`, and `-z`. On BSD, Version 7, PWB, and 32V systems the unary primaries have higher precedence than the binary operators; on System III and System V implementations, the binary operators `=` and `!=` have higher precedence. The change was apparently made for System III so that the construct:

```
test "$1" = "$2"
```

could be made to work even if `$1` started with `-`. It is believed that this change was a mistake because:

— It is not a complete solution; if `$1` expands to `(` or `!`, it still will not work.

— It makes it impossible to use the unary primaries `-n` and `-z` to test for a null string if there is any chance that the string will expand to `=`.

— More importantly, there was the well known workaround of specifying:

```
test X"$1" = X"$2"
```

that always worked.

Unfortunately, when the `=` and `!=` binary primaries were given precedence over the unary primaries, there was no workaround provided for scripts that wanted to reliably specify something like:

```
test -n "$1"
```

because if `$1` expands to `=`, it gives a syntax error.

There was some discussion of outlawing the System V behavior and requiring the more logical precedence that originated in its predecessors and remains in BSD-based systems. However, there are simply too many historical applications that would break if System V were required to make this change; this number dwarfed the number of scripts using combination logic that would then no longer be strictly portable.

POSIX.2 requires that if `test` is called with one, two, three, or four operands it correctly interprets the expression even if there is an alternate syntax tree that could lead to a syntax error. It eliminates the requirement that many string comparisons be protected with leading characters, such as

```
test X"$1" = X"$2"
```

and allows the single-argument *string* form to be used with all possible inputs.

The following examples show some of the changes that are required to be made to make historical BSD and System V-based implementations of `test` conform to this standard:

`test -d =`	POSIX.2	True if there is a directory named =

	BSD	True if there is a directory named =
	System V	Syntax error; = needs two operands
`test -d = -f`	POSIX.2	False
	BSD	Syntax error; it expects `-a` or `-o` after `-d =`
	System V	False

Implementations are prohibited from extending `test` with options because it would make the ‘`test` *string* ’’ case ambiguous for inputs that might match an extended option. Implementations can add primaries and operators, as indicated.

The following options were not included in POSIX.2, although they are provided by some historical implementations, since these facilities and concepts are not supported by POSIX.1 {8}, nor defined in POSIX.2. These operands should not be used by new implementations for other purposes.

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

`-k` *file* True if *file* exists and its sticky bit is set.

`-L` *file* True if *file* is a symbolic link.

`-C` *file* True if *file* is a contiguous file.

`-S` *file* True if *file* is a socket.

`-V` *file* True if *file* is a version file.

The following option was not included because it was undocumented in most implementations, has been removed from some implementations (including System V), and the functionality is provided by the shell (see 3.6.2).

`-l` *string* The length of the string *string* .

The `-b`, `-c`, `-g`, `-p`, `-u`, and `-x` operands are derived from the *SVID* ; historical BSD does not provide them. The `-k` operand is derived from System V; historical BSD does not provide it.

On historical BSD systems, `test -w` *directory* always returned false because `test` tried to open the directory for writing, which always fails.

Some additional primaries newly invented or from the KornShell appeared in an earlier draft as part of the Conditional Command (`[[ ]]`): *s1* > *s2*, *s1* < *s2*, *str* = *pattern*, *str* != *pattern*, *f1* `-nt` *f2*, *f1* `-ot` *f2*, and *f1* `-ef` *f2*. They were not carried forward into the `test` utility when the Conditional Command was removed from the shell because they have not been included in the `test` utility built into historical implementations of the `sh` utility.

The `-t` *file_descriptor* primary is shown with a mandatory argument because the grammar is ambiguous if it can be omitted. Historical implementations have allowed it to be omitted, providing a default of 1.

4.63 `touch` — Change file access and modification times

4.63.1 Synopsis

`touch [−acm][ −r` *ref_file* **|** `−t` *time* `]` *file* . . .

Obsolescent Version :

`touch [−acm][`*date_time* `]` *file* . . .

4.63.2 Description

The `touch` utility shall change the modification and/or access times 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 POSIX.1 {8}; 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 field(s) of the file referenced by the `−r` *ref_file* option-argument, or the *date_time* operand, as specified in the following subclauses. If none of these are specified, `touch` shall use the current time [the value returned by the equivalent of the POSIX.1 {8} *time* () function].

For each *file* operand, `touch` shall perform actions equivalent to the following functions defined in POSIX.1 {8}:

(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 4.63.3.

4.63.3 Options

The `touch` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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 shall be a decimal number of the form:

[[*CC*]*YY*]*MMDD hhmm*[.*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* shall be optional. If neither is given, the current year shall be assumed. If *YY* is specified, but *CC* is not, *CC* shall be derived as follows:

If *YY* is:	*CC* becomes:
69-99	19
00-68	20

The resulting time shall be affected by the value of the **TZ** environment variable. If the resulting time value precedes the Epoch, `touch` shall exit immediately with an error status. The range of valid times past the Epoch is implementation defined, but shall extend to at least midnight 1 January 2000 UTC.

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 shall 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` shall behave as if both the `-a` and `-m` options were specified.

4.63.4 Operands

The following operands shall be supported by the implementation:

file A pathname of a file whose times are to be modified.

date_time (Obsolescent.) Use the specified *date_time* instead of the current time. The operand is a decimal number of the form:

MMDD hhmm[*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 shall be used. If *yy* is in the range 69-99, the year 1969-1999, respectively, shall 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 shall be assumed to be a *date_time* operand. Otherwise, the first operand shall be assumed to be a *file* operand.

4.63.5 External Influences

4.63.5.1 Standard Input

None.

4.63.5.2 Input Files

None.

4.63.5.3 Environment Variables

The following environment variables shall affect the execution of `touch`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

TZ If the *time* option-argument (or operand; see above) is specified, **TZ** shall be used to interpret the time for the specified time zone.

4.63.5.4 Asynchronous Events

Default.

4.63.6 External Effects

4.63.6.1 Standard Output

None.

4.63.6.2 Standard Error

Used only for diagnostic messages.

4.63.6.3 Output Files

None.

4.63.7 Extended Description

None.

4.63.8 Exit Status

The `touch` utility shall exit with one of the following values:

0 The utility executed successfully and all requested changes were made.

>0 An error occurred.

4.63.9 Consequences of Errors

Default.

4.63.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The functionality of `touch` is described almost entirely through references to functions in POSIX.1 {8}. In this way, there is no duplication of effort required for describing such side effects as the relationship of user IDs to the user database, permissions, etc.

The interpretation of time is taken to be ‘‘seconds since the Epoch,’’ as defined by 2.2.2.129. It should be noted that POSIX.1 {8} conforming implementations do not take leap seconds into account when computing seconds since the Epoch. When *SS*=60 is used on POSIX.1 {8} conforming implementations, the resulting time always refers to 1 plus ‘‘seconds since the Epoch’’ for a time when *SS*=59.

Note that 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). 1
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.

History of Decisions Made

There are some significant differences between this `touch` and those in System V and BSD systems. They are upward compatible for existing applications from both implementations.

(1) In System V, an ambiguity exists when a pathname that is a decimal number leads the operands; it is treated as a time value. In BSD, no *time* value is allowed; files may only be `touch`ed to the current time. The [`-t` *time*] construct solves these problems for future portable applications (note that the `-t` option is not existing practice).

(2) The inclusion of the century digits, *CC*, is also new. Note that a ten-digit *time* value is treated as if *YY*, and not *CC*, were specified. The caveat about the range of dates following the Epoch was included as recognition that some UNIX systems will not be able to represent dates beyond the January 18, 2038, because they use *signed int* as a time holder.

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.

The `-r` option was added because several comments requested this capability. This option was named `-f` in an earlier draft, but was changed because the `-f` option is used in the BSD version of `touch` with a different meaning.

At least one historical implementation of touch incremented the exit code if `-c` was specified and the file did not exist. This standard requires exit status zero if no errors occur.

4.64 `tr` — Translate characters

4.64.1 Synopsis

`tr [-cs]` *string1 string2*

`tr -s [-c]` *string1*

`tr -d [-c]` *string1*

`tr -ds [-c]` *string1 string2*

4.64.2 Description

The `tr` utility shall copy the standard input to the standard output with substitution or deletion of selected characters. The options specified and the *string1* and *string2* operands shall control translations that occur while copying characters and collating elements.

4.64.3 Options

The `tr` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`-c` Complement the set of characters specified by *string1* . See 4.64.7.

`-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 4.64.7.

4.64.4 Operands

The following operands shall be supported by the implementation:

string1
string2 Translation control strings. Each string shall represent 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 4.64.7.

4.64.5 External Influences

4.64.5.1 Standard Input

The standard input can be any type of file.

4.64.5.2 Input Files

None.

4.64.5.3 Environment Variables

The following environment variables shall affect the execution of `tr`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the behavior of range expressions and equivalence classes.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments) and the behavior of character classes.

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

4.64.5.4 Asynchronous Events

Default.

4.64.6 External Effects

4.64.6.1 Standard Output

The `tr` output shall be identical to the input, with the exception of the specified transformations.

4.64.6.2 Standard Error

Used only for diagnostic messages.

4.64.6.3 Output Files

None.

4.64.7 Extended Description

The operands *string1* and *string2* (if specified) define two arrays of characters or collating elements. The following conventions can be used to specify characters or collating elements:

character Any character not described by one of the conventions below shall represent itself.

octal Octal sequences can be used to represent characters with specific coded values. An octal sequence shall consist of a backslash followed by the longest sequence of one-, two-, or three-octal-digit characters (01234567). The sequence shall cause the character whose encoding is represented by the one-,
two-, or three-digit octal integer to be placed into the array. If 1
the size of a byte on the system is greater than nine bits, the 1
valid escape sequence used to represent a byte is 1
implementation-defined. Multibyte characters require multi- 1
ple, concatenated escape sequences of this type, including the 1
leading \ for each byte. 1

character The backslash-escape sequences in Table 2-15 (see 2.12) shall be 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 2
endpoints, inclusive, as defined by the current setting of the LC_COLLATE locale category. The starting endpoint shall precede the second endpoint in the current collation order. The characters or collating elements in the range shall be placed in the array in ascending collation sequence. No multicharacter collating elements shall be included in the range.

`[:`*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 shall be accepted when specified in *string1* :

```
alnum   cntrl   lower   space
alpha   digit   print   upper
blank   graph   punct   xdigit
```

When the `-d` and `-s` options are specified together, any of the character class names shall be accepted in *string2* . Otherwise, only character class names `lower` or `upper` shall be accepted 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 shall be interpreted as a request for case conversion. When `[:lower:]` appears in *string1* and `[:upper:]` appears in *string2,* the arrays shall 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 shall contain the characters from the `tolower` mapping in the LC_CTYPE category of the current locale. The first character from each mapping pair shall be in the array for *string1* and the second character from each mapping pair shall be in the array for *string2* in the same relative position.

Except for case conversion, the characters specified by a character class expression shall be 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 behavior 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 shall be 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 shall be placed in the array in an unspecified order.

`[`*x*`*`*n*`]` Represents *n* repeated occurrences of the character or collating symbol *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 shall be 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 shall be interpreted as an octal value. Otherwise, it shall be interpreted as a decimal value.

When the −d option is not specified:

— Each input character or collating element found in the array specified by *string1* shall be replaced by the character or collating element 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 without −d, the complement 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—shall be 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 or collating elements found in the array specified by *string1* shall be deleted.

— When the −c option is specified with −d, all characters except those specified by *string1* shall be deleted. The contents of *string2* shall 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) shall be required.

When the −s option is specified, after any deletions or translations have taken place, repeated sequences of the same character shall 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 shall 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 shall contain only those characters defined as the second characters in each of the `toupper` or `tolower` character pairs, as appropriate.

4.64.8 Exit Status

The `tr` utility shall exit with one of the following values:

- 0 All input was processed successfully.
- >0 An error occurred.

4.64.9 Consequences of Errors

Default.

4.64.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

If necessary, *string1* and *string2* can be quoted to avoid pattern matching by the shell.

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
```

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 standard, both the BSD and System V behaviors are allowed, but a conforming application cannot rely on the BSD behavior. 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.

On historical System V systems, a range expression requires enclosing square- 2
brackets, such as: 2

```
tr '[a-z]' '[A-Z]'
```
2

However, BSD-based systems did not require the brackets and this convention is 2
used by POSIX.2 to avoid breaking large numbers of BSD scripts: 2

```
tr a-z A-Z
```
2

The preceding System V script will continue to work because the brackets, treated 2
as regular characters, are translated to themselves. However, any System V 2

script that relied on `a-z` representing the three characters `a`, `-`, and `z` will have to 2
be rewritten as `az-` or `a\-z`. 2

History of Decisions Made

In some earlier drafts, an explicit option, `–n`, was added to disable the historical behavior of stripping NUL characters from the input. It was felt that automatically stripping NUL characters from the input was not correct functionality. However, the removal of `–n` in a later draft does not remove the requirement that `tr` correctly process NUL characters in its input stream. NUL characters can be stripped by using `tr –d '\000'`.

Historical implementations of `tr` differ widely in syntax and behavior. For example, the BSD version has not needed the bracket characters for the repetition sequence. The POSIX.2 `tr` syntax is based more closely on the System V and XPG3 model, while attempting to accommodate historical BSD implementations. In the case of the short *string2* padding, the decision was to unspecify the behavior and preserve System V and XPG scripts, which might find difficulty with the BSD method. The assumption was made that BSD users of `tr` will have to make accommodations to meet the POSIX.2 syntax anyway, and since it is possible to use the repetition sequence to duplicate the desired behavior, whereas there is no simple way to achieve the System V method, this was the correct, if not desirable, approach.

The use of octal values to specify control characters, while having historical precedents, is not portable. The introduction of escape sequences for control characters should provide the necessary portability. It is recognized that this may cause some historical scripts to break.

A previous draft included support for multicharacter collating elements. Several balloters pointed out that, while `tr` does employ some syntactical elements from regular expressions, the aim of `tr` is quite different; ranges, for instance, do not mean the same thing (''any of the chars in the range matches,'' versus ''translate each character in the range to the output counterpart''). As a result, the previously included support for multicharacter collating elements has been removed. What remains are ranges in current collation order (to support, e.g., accented characters), character classes, and equivalence classes.

In XPG3, the `[:class:]` and `[=equiv=]` conventions are shown with double brackets, as in regular expression syntax. Several balloters objected to this, pointing out that `tr` does not implement regular expression principles, just borrows part of the syntax. Consequently, the `[:class:]` and `[=equiv=]` should be regarded as syntactical elements on a par with `[x*n]`, which is not an RE bracket expression.

4.65 `true` — Return true value

4.65.1 Synopsis

`true`

4.65.2 Description

The `true` utility shall return with exit code zero.

4.65.3 Options

None.

4.65.4 Operands

None.

4.65.5 External Influences

4.65.5.1 Standard Input

None.

4.65.5.2 Input Files

None.

4.65.5.3 Environment Variables

None.

4.65.5.4 Asynchronous Events

Default.

4.65.6 External Effects

4.65.6.1 Standard Output

None.

4.65.6.2 Standard Error

None.

4.65.6.3 Output Files

None.

4.65.7 Extended Description

None.

4.65.8 Exit Status

The `true` utility always exits with a value of zero.

4.65.9 Consequences of Errors

Default.

4.65.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `true` utility is typically used in shell scripts. The special built-in utility `:` (see 3.14.2) is sometimes more efficient than `true`.

History of Decisions Made

The `true` utility has been retained in POSIX.2, even though the shell special built-in `:` provides similar functionality, because `true` is widely used in existing scripts and is less cryptic to novice human script readers.

4.66 `tty` — Return user's terminal name

4.66.1 Synopsis

`tty`

Obsolescent Version :

`tty -s`

4.66.2 Description

The `tty` utility shall write to the standard output the name of the terminal that is open as standard input. The name that is used shall be equivalent to the string that would be returned by the POSIX.1 {8} *ttyname*() function.

4.66.3 Options

The `tty` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`-s` (Obsolescent.) Do not write the terminal name. Only the exit status shall be affected by this option. The terminal status shall be determined as if the POSIX.1 {8} *isatty*() function were used.

4.66.4 Operands

None.

4.66.5 External Influences

4.66.5.1 Standard Input

While no input is read from standard input, standard input shall be examined to determine whether or not it is a terminal, and/or to determine the name of the terminal.

4.66.5.2 Input Files

None.

4.66.5.3 Environment Variables

The following environment variables shall affect the execution of `tty`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE For the obsolescent version, this variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.66.5.4 Asynchronous Events

Default.

4.66.6 External Effects

4.66.6.1 Standard Output

If the `-s` option is specified, standard output shall 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 POSIX.1 {8} *ttyname*() shall be written in the following format:

`"%s\n"`, <*terminal name*>

Otherwise, a message shall be written indicating that standard input is not connected to a terminal. In the POSIX Locale, the `tty` utility shall use the format:

```
"not a tty\n"
```

4.66.6.2 Standard Error

Used only for diagnostic messages.

4.66.6.3 Output Files

None.

4.66.7 Extended Description

None.

4.66.8 Exit Status

The `tty` utility shall exit with one of the following values:

- 0 Standard input is a terminal.
- 1 Standard input is not a terminal.
- >1 An error occurred.

4.66.9 Consequences of Errors

Default.

4.66.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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. The `-s` option was made obsolescent because the same functionality is provided by `test -t 0`, but not dropped completely because historical scripts depend on this form.

History of Decisions Made

The definition of `tty` was made more explicit to explain the difference between a tty and a pathname of a tty.

4.67 `umask` — Get or set the file mode creation mask

4.67.1 Synopsis

`umask [−S] [`*mask*`]`

4.67.2 Description

The `umask` utility shall set the file mode creation mask of the current shell execution environment (see 3.12) to the value specified by the *mask* operand. This mask shall affect the initial value of the file permission bits of subsequently created files.

If the *mask* operand is not specified, the `umask` utility shall write to standard output the value of the invoking process's file mode creation mask.

4.67.3 Options

The `umask` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following option shall be supported by the implementation:

`−S` Produce symbolic output.

The default output style is unspecified, but shall be recognized on a subsequent invocation of `umask` on the same system as a *mask* operand to restore the previous file mode creation mask.

4.67.4 Operands

The following operand shall be supported by the implementation:

mask A string specifying the new file mode creation mask. The string is treated in the same way as the *mode* operand described in 4.7.7 (`chmod` Extended Description).

For a *symbolic_mode* value, the new value of the file mode creation mask shall 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 `-` shall be interpreted relative to the current file mode creation mask; `+` shall cause the bits for the indicated permissions to be cleared in the mask; `-` shall 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.

In the obsolescent octal integer form of *mode*, the specified bits shall be set in the file mode creation mask.

The file mode creation mask shall be set to the resulting numeric value.

As in `chmod`, application use of the octal number form for the *mode* values is obsolescent.

The default output of a prior invocation of `umask` on the same system with no operand shall also be recognized as a *mask* operand. The use of an operand obtained in this way is not obsolescent, even if it is an octal number.

4.67.5 External Influences

4.67.5.1 Standard Input

None.

4.67.5.2 Input Files

None.

4.67.5.3 Environment Variables

The following environment variables shall affect the execution of `umask`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.67.5.4 Asynchronous Events

Default.

4.67.6 External Effects

4.67.6.1 Standard Output

When the *mask* operand is not specified, the `umask` utility shall write a message to standard output that can later be used as a `umask` *mask* operand.

If `−S` is specified, the message shall be in the following format:

`"u=%s,g=%s,o=%s\n"`, *<owner permissions>* , *<group permissions>* , *<other permissions>*

where the three values shall be combinations of letters from the set {`r`, `w`, `x`}; the presence of a letter shall indicate that the corresponding bit is clear in the file mode creation mask.

If a *mask* operand is specified, there shall be no output written to standard output.

4.67.6.2 Standard Error

Used only for diagnostic messages.

4.67.6.3 Output Files

None.

4.67.7 Extended Description

None.

4.67.8 Exit Status

The `umask` utility shall exit with one of the following values:

0 The file mode creation mask was successfully changed, or no *mask* operand was supplied.

>0 An error occurred.

4.67.9 Consequences of Errors

Default.

4.67.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Since `umask` 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 1
execution environment, such as one of the following: 1

```
(umask 002)                    1
nohup umask ...                1
find . -exec umask ... \;      1
```

it will not affect the file mode creation mask of the caller's environment. 1

The table mapping octal mode values in 4.7.7 does not require that the symbolic constants have those particular values.

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.

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.

History of Decisions Made

The description of the historical utility was modified to allow it to use the symbolic modes of `chmod`. The `-s` option used in earlier drafts was changed to `-S` because `-s` could be confused with a *symbolic_mode* form of mask referring to the S_ISUID and S_ISGID bits.

The default output style is implementation defined to permit implementors to provide migration to the new symbolic style at the time most appropriate to their users. Earlier drafts of this standard specified an `-o` flag to force octal mode output. This was dropped because the octal mode may not be sufficient to specify all of the information that may be present in the file mode creation mask when more secure file access permission checks are implemented.

It has been suggested that trusted systems developers might appreciate softening the requirement that the mode mask ''affects'' the file access permissions, since it seems access control lists might replace the mode mask to some degree. The wording has been changed to say that it affects the file permission bits, and leaves the details of the behavior of how they affect the file access permissions to the description in POSIX.1 {8}.

4.68 `uname` — Return system name

4.68.1 Synopsis

```
uname [−amnrsv]
```

4.68.2 Description

By default, the `uname` utility shall write the operating system name to standard output. When options are specified, symbols representing one or more system characteristics shall be written to the standard output. The format and contents of the symbols are implementation defined. On systems conforming to POSIX.1 {8}, the symbols written shall be those supported by the POSIX.1 {8} *uname*() function.

4.68.3 Options

The `uname` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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 shall write the operating system name, as if the `−s` option had been specified.

4.68.4 Operands

None.

4.68.5 External Influences

4.68.5.1 Standard Input

None.

4.68.5.2 Input Files

None.

4.68.5.3 Environment Variables

The following environment variables shall affect the execution of `uname`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.68.5.4 Asynchronous Events

Default.

4.68.6 External Effects

4.68.6.1 Standard Output

By default, the output shall be a single line of the following form:

`"%s\n"`, *<sysname>*

If the `-a` option is specified, the output shall be a single line of the following form:

`"%s %s %s %s %s\n"`, *<sysname>*, *<nodename>*, *<release>*, *<version>*, *<machine>*

Additional implementation-defined symbols may be written; all such symbols shall be written at the end of the line of output before the `<newline>`.

If options are specified to select different combinations of the symbols, only those symbols shall be written, in the order shown above for the `−a` option. If a symbol is not selected for writing, its corresponding trailing `<blank>`s also shall not be written.

4.68.6.2 Standard Error

Used only for diagnostic messages.

4.68.6.3 Output Files

None.

4.68.7 Extended Description

None.

4.68.8 Exit Status

The `uname` utility shall exit with one of the following values:

0	The requested information was successfully written.
>0	An error occurred.

4.68.9 Consequences of Errors

Default.

4.68.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The following command:

```
uname −sr
```

writes the operating system name and release level, separated by one or more `<blank>`s.

Note that any of the symbols could include embedded `<space>`s, 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 inter-system communication addressing.

History of Decisions Made

It was suggested that this utility cannot be used portably, since the format of the symbols is implementation defined. The POSIX.1 {8} working group could not achieve consensus on defining these formats in the underlying *uname*() function and there is no expectation that POSIX.2 would be any more successful. In any event, some applications may still find this historical utility of value. For example, the symbols could be used for system log entries or for comparison with operator or user input.

4.69 uniq — Report or filter out repeated lines in a file

4.69.1 Synopsis

`uniq` [−c|−d|−u] [−f *fields*] [−s *chars*] [*input_file* [*output_file*]]

Obsolescent Version:

`uniq` [−c|−d|−u] [−*n*] [+*m*] [*input_file* [*output_file*]]

4.69.2 Description

The `uniq` utility shall 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 shall not be written.

Repeated lines in the input shall not be detected if they are not adjacent.

4.69.3 Options

The `uniq` utility shall conform to the utility argument syntax guidelines described in 2.10.2; 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 shall be supported by the implementation:

−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* shall be a positive decimal integer. A field is the maximal string matched by the basic regular expresssion:

```
[[:blank:]]*[^[:blank:]]*
```

If the *fields* option-argument specifies more fields than appear on an input line, a null string shall be used for comparison.

−s *chars* Ignore the first *chars* characters when doing comparisons, where *chars* shall be a positive decimal integer. If specified in conjunction with the −f option, the first *chars* characters after the first *fields* fields shall be ignored. If the *chars* option-argument specifies more characters than remain on an input line, a null string shall be used for comparison.

−u Suppress the writing of lines that are repeated in the input.

−*n* (Obsolescent.) Equivalent to −f *fields* with *fields* set to *n*.

+*m* (Obsolescent.) Equivalent to −s *chars* with *chars* set to *m*.

4.69.4 Operands

The following operands shall be supported by the implementation:

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 shall be used.

output_file A pathname of the output file. If the *output_file* operand is not specified, the standard output shall be used. The results are unspecified if the file named by *output_file* is the file named by *input_file*.

4.69.5 External Influences

4.69.5.1 Standard Input

The standard input shall be used only if no *input_file* operand is specified or if *input_file* is −. See Input Files.

4.69.5.2 Input Files

The input file shall be a text file.

4.69.5.3 Environment Variables

The following environment variables shall affect the execution of uniq:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and which characters constitute a `<blank>` in the current locale.

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

4.69.5.4 Asynchronous Events

Default.

4.69.6 External Effects

4.69.6.1 Standard Output

The standard output shall be used only if no *output_file* operand is specified. See Output Files.

4.69.6.2 Standard Error

Used only for diagnostic messages.

4.69.6.3 Output Files

If the `−c` option is specified, the output file shall be empty or each line will be of the form:

`"%d %s"`, <*number of duplicates*> , <*line*>

otherwise, the output file will be empty or each line will be of the form:

`"%s"`, <*line*>

4.69.7 Extended Description

None.

4.69.8 Exit Status

The `uniq` utility shall exit with one of the following values:

0 The utility executed successfully.

>0 An error occurred.

4.69.9 Consequences of Errors

Default.

4.69.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Some historical implementations have limited lines to be 1080 bytes in length, which will not meet the implied {LINE_MAX} limit.

The `sort` utility (see 4.58) can be used to cause repeated lines to be adjacent in the input file.

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`'s 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.

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).

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
```

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
```

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.

History of Decisions Made

The `-f` and `-s` options were added to replace the obsolescent `-`*n* and `+`*m* options so that `uniq` could meet the syntax guidelines in an upward-compatible way.

The output specifications in Output Files do not show a terminating `<newline>` because they both specify <*line*>, which includes its own `<newline>` (because of the definition of *line*).

4.70 `wait` — Await process completion

4.70.1 Synopsis

`wait` [*pid* . . .]

4.70.2 Description

When an asynchronous list (see 3.9.3.1) is started by the shell, the process ID of the last command in each element of the asynchronous list shall become known in 1
the current shell execution environment; see 3.12.

If the `wait` utility is invoked with no operands, it shall 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 shall wait until all of them have terminated. If one or more *pid* operands are specified that represent unknown process IDs, `wait` shall treat them as if they were known process IDs that exited with exit status 127. The exit status returned by the `wait` utility shall 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.

4.70.3 Options

None.

4.70.4 Operands

The following operand shall be supported by the implementation:

pid The unsigned decimal integer process ID of a command, for which the utility is to wait for the termination.

4.70.5 External Influences

4.70.5.1 Standard Input

None.

4.70.5.2 Input Files

None.

4.70.5.3 Environment Variables

The following environment variables shall affect the execution of `wait`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

4.70.5.4 Asynchronous Events

Default.

4.70.6 External Effects

4.70.6.1 Standard Output

None.

4.70.6.2 Standard Error

Used only for diagnostic messages.

4.70.6.3 Output Files

None.

4.70.7 Extended Description

None.

4.70.8 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` shall 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 shall be greater than 128 and shall be distinct from the exit status generated by other signals, but the exact value is unspecified. (See the `kill -l` option in 4.32.) Otherwise, the `wait` utility shall 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.

4.70.9 Consequences of Errors

Default.

4.70.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, 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.

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.
```

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 initialization error because the exit status returned was usually zero if the requested process ID was not found. POSIX.2 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.)

This new functionality was added because it is needed to accurately determine the exit status of any asynchronous list. The only compatibility problem that this change creates is for a script like:

```
while sleep 60
do
        job&
        echo Job started $(date) as $!
done
```

which will cause the shell to keep track of all of the jobs started until the script terminates or runs out of memory. This would not be a problem if the loop did not reference `$!` or if the script would occasionally `wait` for jobs it started.

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.

History of Decisions Made

The description of `wait` does not refer to the *waitpid* () function from POSIX.1 {8}, because that would needlessly overspecify this interface. However, the wording requires that `wait` is required to wait for an explicit process when it is given an argument, so that the status information of other processes is not consumed. Historical implementations use POSIX.1 {8} *wait* () until *wait* () returns the requested process ID or finds that the requested process does not exist. Because this means that a shell script could not reliably get the status of all background children if a second background job was ever started before the first job finished, it is recommended that the `wait` utility use a method such as the functionality provided by the *waitpid* () function in POSIX.1 {8}.

The ability to wait for multiple *pid* operands was adopted from the KornShell at the request of ballot comments and objections.

Some implementations of `wait` support waiting for asynchronous lists identified by the use of job identifiers. For example, `wait %1` would wait for the first background job. This standard does not address job control issues, but allows these features to be added as extensions. Job control facilities will be provided by the UPE.

4.71 wc — Word, line, and byte count

4.71.1 Synopsis

wc [−clw] [*file* ...]

4.71.2 Description

The wc utility shall read one or more input files and, by default, write the number of <newline>s, words, and bytes contained in each input file to the standard output.

The utility also shall write a total count for all named files, if more than one input file is specified.

The wc utility shall consider a *word* to be a nonzero-length string of characters delimited by white space.

4.71.3 Options

The wc utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−c Write to the standard output the number of bytes in each input file.

−l Write to the standard output the number of <newline>s in each input file.

−w Write to the standard output the number of words in each input file.

When any option is specified, wc shall report only the information requested by the specified option(s).

4.71.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If no *file* operands are specified, the standard input shall be used.

4.71.5 External Influences

4.71.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

4.71.5.2 Input Files

The input files may be of any type.

4.71.5.3 Environment Variables

The following environment variables shall affect the execution of `wc`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and which characters are defined as ‘‘white space’’ characters.

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

4.71.5.4 Asynchronous Events

Default.

4.71.6 External Effects

4.71.6.1 Standard Output

By default, the standard output shall contain a line for each input file of the form:

`"%d %d %d %s\n"`, *<newlines>* , *<words>* , *<bytes>*, *<file>*

If any options are specified and the `-l` option is not specified, the number of `<newline>`s shall not be written.

If any options are specified and the `-w` option is not specified, the number of words shall not be written.

If any options are specified and the `-c` option is not specified, the number of bytes shall not be written.

If no input *file* operands are specified, no name shall be written and no `<blank>`s preceding the pathname shall be written.

If more than one input *file* operand is specified, an additional line shall be written, of the same format as the other lines, except that the word `total` (in the POSIX Locale) shall be written instead of a pathname and the total of each column shall be written as appropriate. Such an additional line, if any, shall be written at the end of the output.

4.71.6.2 Standard Error

Used only for diagnostic messages.

4.71.6.3 Output Files

None.

4.71.7 Extended Description

None.

4.71.8 Exit Status

The `wc` utility shall exit with one of the following values:

0	Successful completion.
>0	An error occurred.

4.71.9 Consequences of Errors

Default.

4.71.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

The output file format pseudo- *printf* () string was derived from the HP-UX version of `wc`; the System V version:

```
"%7d%7d%7d %s\n"
```

produces possibly ambiguous and unparsable results for very large files, as it assumes no number will exceed six digits.

Some historical implementations use only `<space>`, `<tab>`, and `<newline>` as word separators. The equivalent of the C Standard {7} *isspace* () function is more appropriate.

The `−c` option stands for ‘‘character’’ count, even though it counts bytes. This stems from the sometimes erroneous historical view that bytes and characters are the same size.

Earlier drafts only specified the results when input files were text files. The current specification more closely matches existing practice. (Bytes, words, and `<newline>`s are counted separately and the results are written when an end-of-file is detected.)

Historical implementations of the `wc` utility only accepted one argument to specify the options `−c`, `−l`, and `−w`. Some of them also had multiple occurrences of an option cause the corresponding count to be output multiple times and having the order of specification of the options affect the order of the fields on output, but did not document either of these. Because common usage either specifies no options or only one option and because none of this was documented, the changes required by this standard should not break many existing applications (and does not break any historical portable applications.)

4.72 xargs — Construct argument list(s) and invoke utility

4.72.1 Synopsis

xargs [−t] [−n *number* [−x]] [−s *size*] [*utility* [*argument* . . .]]

4.72.2 Description

The xargs utility shall construct 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 shall then invoke the constructed command line and wait for its completion. This sequence shall be 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. 1

Arguments in the standard input shall be separated by unquoted <blank>s, or unescaped <blank>s or <newline>s. A string of zero or more nondouble-quote (") and non-<newline> characters can be quoted by enclosing them in double-quotes. A string of zero or more nonapostrophe (') 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* shall 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 shall 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 shall 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 POSIX.1 {8} 3.1.2) shall not exceed {ARG_MAX}−2048 bytes. Within this constraint, if neither the −n nor the −s option is specified, the default command line length shall be at least {LINE_MAX}.

4.72.3 Options

The xargs utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−n *number* Invoke *utility* using as many standard input arguments as possible, up to *number* (a positive decimal integer) arguments maximum. Fewer arguments shall 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.

`−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 shall be used if:

— The total number of arguments exceeds that specified by the `−n` option, or

— End of file is encountered on standard input before *size* bytes are accumulated.

Implementations shall support values of *size* up to at least {LINE_MAX} bytes, provided that the constraints specified in 4.72.2 are met. It shall not be considered an error if a value larger than that supported by the implementation or exceeding the constraints specified in 4.72.2 is given; `xargs` shall use the largest value it supports within the constraints.

`−t` Enable trace mode. Each generated command line shall be written to standard error just prior to invocation.

`−x` Terminate if a command line containing *number* arguments (see the `−n` option above) will not fit in the implied or specified size (see the `−s` option above).

4.72.4 Operands

The following operands shall be supported by the implementation:

utility The name of the utility to be invoked, found by search path using the **PATH** environment variable, described in 2.6. If *utility* is omitted, the default shall be the `echo` utility (see 4.19). If the *utility* operand names any of the special built-in utilities in 3.14, the results are undefined.

argument An initial option or operand for the invocation of *utility* .

4.72.5 External Influences

4.72.5.1 Standard Input

The standard input shall be a text file. The results are unspecified if an end-of-file condition is detected immediately following an escaped `<newline>`.

4.72.5.2 Input Files

None.

4.72.5.3 Environment Variables

The following environment variables shall affect the execution of `xargs`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

4.72.5.4 Asynchronous Events

Default.

4.72.6 External Effects

Any external effects are a result of the invocation of the utility *utility* , in a manner specified by that utility.

4.72.6.1 Standard Output

None.

4.72.6.2 Standard Error

Used for diagnostic messages and the `−t` option. If the `−t` option is specified, the *utility* and its constructed argument list shall be written to standard error, as it will be invoked, prior to invocation.

4.72.6.3 Output Files

None.

4.72.7 Extended Description

None.

4.72.8 Exit Status

The `xargs` utility shall exit with one of the following values:

- 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 nonzero exit status, or some other error occurred. 1
- 126 The utility specified by *utility* was found but could not be invoked. 1
- 127 The utility specified by *utility* could not be found. 1

4.72.9 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 shall write a diagnostic message and exit without processing any remaining input.

4.72.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `xargs` utility is usually found only in System V-based systems; BSD systems provide an `apply` utility that provides functionality similar to `xargs –n` *number* . The *SVID* lists `xargs` as a software development extension; POSIX.2 does not share the view that it is used only for development, and therefore it is not optional.

Note that input is parsed as lines and `<blank>`s 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 file names contain any `<blank>`s or `<newline>`s. 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.

The following command will combine the output of the parenthesized commands onto one line, which is then written to the end of file `log`:

```
(logname; date; printf "%s\n" "$0 $*") | xargs >>log
```

The following command will invoke `diff` with successive pairs of arguments originally typed as command line arguments (assuming there are no embedded `<blank>`s in the elements of the original argument list):

```
printf "%s\n" "$*" | xargs -n 2 -x diff
```

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.

History of Decisions Made

The list of options has been scaled down extensively. As it had stood, the `xargs` utility did not exhibit an economy of powerful, modular, or extensible functionality.

The classic application of the `xargs` utility is in conjunction with the `find` utility to reduce the number of processes launched by a simplistic use of the `find` `−exec` combination. The `xargs` utility is also used to enforce an upper limit on memory required to launch a process. With this basis in mind, POSIX.2 selected only the minimal features required.

The `−n` *number* option was classically used to evoke a utility using pairs of operands, yet the general case has problems when *utility* spawns child processes of its own. The `xargs` utility can sap resources from these children, especially those sharing the parent's environment.

The `command`, `env`, `nohup`, 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.

Although the 255 exit status is mostly an accident of historical implementations, it 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. Any nonzero exit status from a utility will fall into the 1−125 range when `xargs` exits. There is no statement of how the various nonzero utility exit status codes are accumulated by `xargs`. The value could be the addition of all codes, their highest value, the last one received, or a single value such as 1. Since no algorithm is arguably better than the others, and since many of the POSIX.2 standard utilities say little more (portably) than ‘‘pass/fail,’’ no new algorithm was invented. 1

Several other `xargs` options were withdrawn because simple alternatives already exist within the standard. For example, the `−e`*eofstr* option has a `sed` work around. The `−i`*replstr* option can be just as efficiently performed using a shell `for` loop. Since `xargs` will *exec*() with each input line, the `−i` option will usually not exploit `xarg`’s grouping capabilities.

The `−s` option was reinstated since many of the balloters on Draft 8 felt that it was preferable to the `−r` option invented for that draft that required the implementation to use {ARG_MAX} − *size* bytes for command lines.

The requirement that `xargs` never produce command lines such that invocation of *utility* is within 2048 bytes of hitting the POSIX.1 {8} *exec* {ARG_MAX} limitations is intended to guarantee that the invoked utility has a little bit of room to modify its environment variables and command line arguments and still be able to invoke another utility. Note that the minimum {ARG_MAX} allowed by POSIX.1 {8} is 4096 and the minimum value allowed by POSIX.2 is 2048; therefore, the 2048-byte difference seems reasonable. Note, however, that `xargs` may never be able to invoke a utility if the environment passed in to `xargs` comes close to using {ARG_MAX} bytes.

The version of `xargs` required by POSIX.2 is required to wait for the completion of the invoked command before invoking another command. This was done because existing scripts using `xargs` assumed sequential execution. Implementations wanting to provide parallel operation of the invoked utilities are encouraged to add an option enabling parallel invocation, but should still wait for termination of all of the children before `xargs` terminates normally.

Section 5: User Portability Utilities Option

Editor's Note: This empty section is placeholder for a future revision (the User Portability Extension, P1003.2a) to contain descriptions of utilities that are suitable for user portability on asynchronous character terminals. P1003.2a is currently balloting within the IEEE. Contact the IEEE Standards Office to obtain a copy of the latest draft.

Section 6: Software Development Utilities Option

This section describes utilities used for the development of applications, including compilation or translation of source code, the creation and maintenance of library archives, and the maintenance of groups of interdependent programs.

The utilities described in this section may be provided by the conforming system; however, any system claiming conformance to the **Software Development Utilities Option** shall provide all of the utilities described here.

6.1 `ar` — Create and maintain library archives

6.1.1 Synopsis

`ar −d [−v]` *archive file* ...

`ar −p [−v]` *archive* [*file* ...]

`ar −r [−cuv]` *archive file* ...

`ar −t [−v]` *archive* [*file* ...]

`ar −x [−v]` *archive* [*file* ...]

6.1.2 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 shall format the archive so that it is usable as a library for link editing (see A.1 and C.2). When some of the archived files are not valid object files, the suitability of the archive for library use is undefined.

All *file* operands can be pathnames. However, files within archives shall be 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 shall be 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 such files, however, each *file* operand shall match only the first archive file having a name that is the same as the last component of the *file* operand.

6.1.3 Options

The `ar` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−c` Suppress the diagnostic message that is written to standard error by default when the archive file *archive* is created.

`−d` Delete *file*(s) from *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 shall be written in the order of the archive.

`−r` Replace or add *file*(s) to *archive*. If the archive named by *archive* does not exist, a new archive file shall be created and a diagnostic message shall 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 shall not change the order of the archive. Files that do not replace existing files shall be appended to the archive.

`−t` Write a table of contents of *archive* to the standard output. The files specified by the *file* operands shall be included in the written list. If no *file* operands are specified, all files in *archive* shall be included in the order of the archive.

`−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 6.1.6.1.

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 6.1.6.1.

When used with `−t`, include a long listing of information about the files within the archive, as described in 6.1.6.1.

`−x` Extract the files named by the *file* operands from *archive*. The contents of the archive file shall not be changed. If no *file* operands are given, all files in the archive shall 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 shall be set to the time the file is extracted from the archive.

6.1.4 Operands

The following operands shall be supported by the implementation:

archive A pathname of the archive file.

file A pathname. Only the last component shall 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 shall not truncate valid filenames of files added to, or replaced in, the archive.

6.1.5 External Influences

6.1.5.1 Standard Input

None.

6.1.5.2 Input Files

The input file named by *archive* shall be a file in the format created by `ar -r`.

6.1.5.3 Environment Variables

The following environment variables shall affect the execution of `ar`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

LC_TIME This variable shall determine the format and content for date and time strings written by `ar`.

6.1.5.4 Asynchronous Events

Default.

6.1.6 External Effects

6.1.6.1 Standard Output

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` shall 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 shall be 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> shall be formatted the same as the *<file mode>* string defined in 4.39.6.1 (Standard Output 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>
shall be equivalent to the `%b` format in `date` (see 4.15).

<day-of-month> shall be equivalent to the `%e` format in `date`.

<hour> shall be equivalent to the `%H` format in `date`.

<minute> shall be equivalent to the `%M` format in `date`.

<year> shall be 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.

6.1.6.2 Standard Error

Used only for diagnostic messages. The diagnostic message about creating a new archive when `−c` is not specified shall not modify the exit status.

6.1.6.3 Output Files

Archives are files with unspecified formats.

6.1.7 Extended Description

None.

6.1.8 Exit Status

The `ar` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

6.1.9 Consequences of Errors

Default.

6.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The archive format is not described. It is recognized that there are several known `ar` formats, which are not compatible. The `ar` utility is being included, however, to allow creation of archives that are intended for use only on the same machine. The archive file is specified as a file and it can be moved as a file. This does allow an archive to be moved from one machine to another machine that uses the same implementation of `ar`.

Utilities such as `pax` (and its forebears `tar` and `cpio`) also provide portable 1
''archives.'' This is a not a duplication; the `ar` interface is included in the standard to provide an interface primarily for `make` and the compilers, based on a historical model.

In historical implementations, the `−q` option is known to execute quickly because `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. The remarks may or may not hold true for a brand-new POSIX.2 implementation; and hence, these remarks have been moved out of the specification and into the Rationale.

Likewise, historical implementations maintain a symbol table to speed searches, particularly when the archive contains object files. However, future implementors may or may not use a symbol table, and the `−s` option was removed from this clause to permit implementors freedom of choice. Instead, the requirement that archive libraries be suitable for link editing was added to ensure the intended functionality. Systems such as System V maintain the symbol table without requiring the use of `−s`, so adding `−s` (even if it were worded as allowing a no-op) would essentially require all portable applications to use it in all invocations involving libraries.

The Operands subclause requires what might seem to be true without specifying it: the archive cannot truncate the filenames below {NAME_MAX}. Some historical implementations do so, however, causing unexpected results for the application. Therefore, POSIX.2 makes the requirement explicit to avoid misunderstandings.

According to the System V documentation, the options `-dmpqrtx` are not required to begin with a hyphen (−). POSIX.2 requires that a conforming application use the leading hyphen.

When extracting files with long filenames into a file system that supports only shorter filenames, an undefined condition occurs. Typical implementation actions might be one of the following:

— Extract and truncate the filename only when an existing file would not be overlaid.

— Extract and truncate the filename and overlay an existing file only if some extension such as another command-line option were used to override this safety feature.

— Refuse to extract any files unless an extension overrode the default.

The archive format used by the 4.4BSD implementation is documented in the rationale as an example:

> A file created by `ar` begins with the ‘‘magic’’ string ‘‘`!<arch>\n`’’. The rest of the archive is made up of objects, each of which is composed of a header for a file, a possible filename, and the file contents. The header is portable between machine architectures, and, if the file contents are printable, the archive is itself printable.
>
> The header is made up of six ASCII fields, followed by a two-character trailer. The fields are the object name (16 characters), the file last modification time (12 characters), the user and group IDs (each 6 characters), the file mode (8 characters) and the file size (10 characters). All numeric fields are in decimal, except for the file mode, which is in octal.
>
> The modification time is the file *st_mtime* field. The user and group IDs are the file *st_uid* and *st_gid* fields. The file mode is the file *st_mode* field. The file size is the file *st_size* field. The two-byte trailer is the string ‘‘`` ` ``<newline>’’.
>
> Only the name field has any provision for overflow. If any filename is more than 16 characters in length or contains an embedded space, the string ‘`#1/`’’ followed by the ASCII length of the name is written in the name field. The file size (stored in the archive header) is incremented by the length of the name. The name is then written immediately following the archive header.
>
> Any unused characters in any of these fields are written as <space> characters. If any fields are their particular maximum number of characters in length, there will be no separation between the fields.
>
> Objects in the archive are always an even number of bytes long; files that are an odd number of bytes long are padded with a <newline> character, although the size in the header does not reflect this.

History of Decisions Made

The `ar` utility description requires that (when all its members are valid object files) `ar` produce an object code library, which the linkage editor can use to extract object modules. If the linkage editor needs a symbol table to permit random access to the archive, `ar` must provide it; however, `ar` does not require a symbol table. The historical `−m` and `−q` positioning options were omitted, as were the positioning modifiers formerly associated with the `−m` and `−r` options, because the two functions of positioning are handled by the `ranlib`-style (a utility found on some historical systems to create symbol tables within the archive) symbol tables and/or the ability of portable applications to create multiple archives instead of loading from a single archive.

Earlier drafts had elaborate descriptions in the Asynchronous Events subclause about how signals were caught and then resent to itself. These were removed in favor of the default case because they are essentially implementation details, unnecessary for the application. Similarly, information about where (and if) temporary files are created was removed from earlier drafts.

The BSD `−o` option was omitted. It is a rare portable application that will use `ar` to extract object code from a library with concern for its modification time, since this can only be of importance to `make`. Hence, since this functionality is not deemed important for applications portability, the modification time of the extracted files is set to the current time.

There is at least one known implementation (for a small computer) that can accommodate only object files for that system, disallowing mixed object and other files. The ability to handle any type of file is not only existing practice for most implementations, but is also a reasonable expectation.

Consideration was given to changing the output format of `ar −tv` to the same format as the output of `ls −l`. This would have made parsing the output of `ar` the same as that of `ls`. This was rejected in part because the current `ar` format is commonly used and changes would break existing usage. Second, `ar` gives the user ID and group ID in numeric format separated by a slash. Changing this to be the user name and group name would not be right if the archive were moved to a machine that contained a different user database. Since `ar` cannot know whether the archive file was generated on the same machine, it cannot tell what to report.

The text on the `−ur` option combination is historical practice—since one filename can easily represent two different files (e.g., `/a/foo` and `/b/foo`), it is reasonable to replace the member in the archive even when the modification time in the archive is identical to that in the file system.

6.2 make — Maintain, update, and regenerate groups of programs

6.2.1 Synopsis

```
make [−einpqrst][−f makefile ]... [ −k | −S ][macro=name ]...
     [target_name ...]
```

6.2.2 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, shall consist of a target, optional prerequisites, and optional commands to be executed when a prerequisite is newer than the target. There are two types of rules:

— 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 shall have a collection of built-in macros and inference rules that infer prerequisite relationships to simplify maintenance of programs.

To receive exactly the behavior described in this clause, a portable makefile shall:

— Include the special target .POSIX (see 6.2.7.3)

— Omit any special target reserved for implementations (a leading period followed by uppercase letters) that has not been specified by this clause.

The behavior of make is unspecified if either or both of these conditions are not met.

6.2.3 Options

The make utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−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 `"-"` shall denote the standard input. There can be multiple instances of this option, and they shall be processed in the order specified. The effect of specifying the same option-argument more than once is unspecified. See 6.2.7.1.

`-i` Ignore error codes returned by invoked commands. This mode is the same as if the special target `.IGNORE` were specified without prerequisites. See 6.2.7.2.

`-k` Continue to update other targets that do not depend on the current target if a nonignored 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 shall be executed. In this mode, lines with an at-sign (`@`) character prefix shall 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 shall not be updated if this option is specified. However, a command line (associated with the targets) with a plus-sign (`+`) prefix shall 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 shall 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 shall be the same as if the special target `.SILENT` were specified without prerequisites. See 6.2.7.2.

`-t` Update the modification time of each target as though a `touch` *target* had been executed. See `touch` in 4.63. Targets that have prerequisites but no commands (see 6.2.7.3), or that are already up-to-date, shall 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 shall 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 shall take precedence. The **MAKEFLAGS** environment variable shall be evaluated first and the command line shall be evaluated second. Assignments to the `MAKEFLAGS` macro shall be evaluated as described in 6.2.5.3.

6.2.4 Operands

The following operands shall be supported by the implementation:

target_name Target names, as defined in 6.2.7. 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 shall be used.

macro=name Macro definitions, as defined in 6.2.7.4.

If the *target_name* and *macro=name* operands are intermixed on the command line, the results are unspecified.

6.2.5 External Influences

6.2.5.1 Standard Input

The standard input shall be used only if the *makefile* option-argument is −. See Input Files.

6.2.5.2 Input Files

The input file, otherwise known as the makefile, is a text file containing rules, 1
macro definitions, and comments. (See 6.2.7.) 1

6.2.5.3 Environment Variables

The following environment variables shall affect the execution of `make`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

MAKEFLAGS This variable shall be interpreted as a character string representing a series of option characters to be used as the default options. The implementation shall 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>` 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>`-separated as described in 2.10.2. 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 shall not be subjected to the word expansions (see 3.6) associated with parsing the command line values.

When the command-line options `−f` or `−p` are used, they shall take effect regardless of whether they also appear in **MAKEFLAGS**. If they otherwise appear in **MAKEFLAGS**, the result is undefined.

The **MAKEFLAGS** variable shall 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` shall be added to the `MAKEFLAGS` macro. The `MAKEFLAGS` macro shall be passed into the environment as an environment variable for all child processes. If the `MAKEFLAGS` macro is subsequently set by the makefile, it shall replace the **MAKEFLAGS** variable currently found in the environment.

The value of the **SHELL** environment variable shall not be used as a macro and shall 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, shall be used as macros, as defined in 6.2.7.4.

6.2.5.4 Asynchronous Events

If not already ignored, `make` shall 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 shall be reported in diagnostic messages of unspecified format, written to standard error. After this cleanup process, if any, `make` shall take the standard action for all other signals; see 2.11.5.4.

6.2.6 External Effects

6.2.6.1 Standard Output

The `make` utility shall 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 shall write a message to standard output indicating that no action was taken.

6.2.6.2 Standard Error

Used only for diagnostic messages.

6.2.6.3 Output Files

None. However, utilities invoked by `make` may create additional files.

6.2.7 Extended Description

The `make` utility attempts to perform the actions required to ensure that the specified target(s) 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 shall treat all prerequisites as targets themselves and recursively ensure that they are up-to-date, processing them in the order in which they appear in the rule. The `make` utility shall use the modification times of files to determine if the corresponding targets are out-of-date. (See 2.9.1.6.)

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 shall be executed. If there are no commands listed for the target, the target shall be treated as up-to-date.

6.2.7.1 Makefile Syntax

A makefile can contain rules, macro definitions (see 6.2.7.4), and comments. There are two kinds of rules: inference rules (6.2.7.5) and target rules (6.2.7.3). The `make` utility shall contain a set of built-in inference rules. If the `−r` option is present, the built-in rules shall not be used and the suffix list shall be 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 shall be that of the last one specified. Comments start with a number-sign (#) and continue until an unescaped `<newline>` is reached.

By default, the file `./makefile` shall be used. If `./makefile` is not found, the file `./Makefile` shall be tried. If neither `./makefile` nor `./Makefile` are found, other implementation-defined pathnames may also be tried.

The `−f` option shall direct `make` to ignore `./makefile` and `./Makefile` (and any implementation-defined variants) and use the specified argument as a makefile instead. If the − argument is specified, standard input shall be used.

The term *makefile* is used to refer to any rules provided by the user whether in `./makefile`, `./Makefile`, or specified by the `−f` option.

The rules in makefiles shall consist of the following types of lines: target rules, including special targets (see 6.2.7.3); inference rules (see 6.2.7.5); macro definitions (see 6.2.7.4); empty lines; and comments. Comments start with a number sign (#) and continue until an unescaped <newline> is reached.

When an escaped <newline> (one preceded by a backslash) is found anywhere in the makefile, it shall be replaced, along with any leading white space on the following line, with a single <space>.

6.2.7.2 Makefile Execution

Command lines shall be 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 command(s) in the line. A <tab> character may precede the command to standard output. Commands shall be executed by passing the command line to the command interpreter in the same manner as if the string were the argument to the function in 7.1.1 [such as the *system* () function in the C binding].

The environment for the command being executed shall contain all of the variables in the environment of `make`. The macros from the command line to `make` shall be added to `make`'s environment. Other implementation-defined variables may also be added to `make`'s environment. If any command-line macro has been defined elsewhere, the command-line value shall 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 shall be created by `make`, and shall contain all options specified on the command line except for the `−f` and `−p` options. It may also contain implementation-defined options.

By default, when `make` receives a nonzero 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 shall not be 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 shall 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 shall not be written to standard output before it is executed.

+ If the command prefix contains a plus-sign, this indicates a command line that shall be executed even if `-n`, `-q`, or `-t` is specified.

6.2.7.3 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>-separated, nonnull list of targets, then a colon, then a <blank>-separated, possibly empty list of prerequisites. Text following a semicolon, if any, and all following lines that begin with a <tab>, are command lines to be executed to update the target. The first nonempty line that does not begin with a <tab> or # shall begin a new entry. An empty or blank line, or a line beginning with #, may begin a new entry.

Applications shall select target names from the set of characters consisting solely of periods, underscores, digits, and alphabetics from the portable character set (see 2.4). Implementations may allow other characters in target names as extensions. The interpretation of targets containing the characters ‘%’’ and ‘‘"’’ is implementation defined.

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 shall be specified with commands, but without prerequisites. The commands shall 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 shall cause errors from commands associated with them to be ignored in the same manner as specified by the `-i` option. Subsequent occurrences of `.IGNORE` shall add to the list of targets ignoring command errors. If no prerequisites are specified, `make` shall behave as if the `-i` option had been specified and errors from all commands associated with all targets shall be ignored.

`.POSIX` This special target shall be specified without prerequisites or commands. If it appears before the first noncomment line in the makefile, `make` shall process the makefile as specified by this clause; otherwise, the behavior of `make` is unspecified.

`.PRECIOUS` Prerequisites of this special target shall not be removed if `make` receives one of the asynchronous events explicitly described in 6.2.5.4. Subsequent occurrences of `.PRECIOUS` shall add to the list of precious files. If no prerequisites are specified, all targets in the makefile shall be treated as if specified with `.PRECIOUS`.

`.SILENT` Prerequisites of this special target are targets themselves; this shall cause commands associated with them to not be written to the standard output before they are executed. Subsequent occurrences of `.SILENT` shall add to the list of targets with silent commands. If no prerequisites are specified, `make` shall behave as if the `−s` option had been specified and no commands or touch messages associated with any target shall be written to standard output.

`.SUFFIXES` Prerequisites of `.SUFFIXES` shall be appended to the list of known suffixes and are used in conjunction with the inference rules (see 6.2.7.5). If `.SUFFIXES` does not have any prerequisites, the list of known suffixes shall be cleared. Makefiles shall not associate commands with `.SUFFIXES`.

Targets with names consisting of a leading period followed by the uppercase letters `POSIX` and then any other characters are reserved for future standardization. Targets with names consisting of a leading period followed by one or more uppercase letters are reserved for implementation extensions.

6.2.7.4 Macros

Macro definitions are in the form:

string1 = [*string2*] 1

The macro named *string1* is defined as having the value of *string2* , where *string2* is defined as all characters, if any, after the equals-sign, up to a comment charac- 1
ter (#) or an unescaped `<newline>` character. Any `<blank>`s immediately before or after the equals-sign shall be ignored.

Subsequent appearances of `$(`*string1*`)` or `${`*string1*`}` shall be replaced by *string2* . The parentheses or braces are optional if *string1* is a single character. The macro `$$` shall be replaced by the single character `$`.

Applications shall select macro names from the set of characters consisting solely 2
of periods, underscores, digits, and alphabetics from the portable character set 2
(see 2.4). A macro name shall not contain an equals-sign. Implementations may 2
allow other characters in macro names as extensions. 2

Macros can appear anywhere in the makefile. Macros in target lines shall be evaluated when the target line is read. Macros in command lines shall be evaluated when the command is executed. Macros in macro definition lines shall not be evaluated until the new macro being defined is used in a rule or command. A macro that has not been defined shall 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 2
is performed. The *subst1* to be replaced shall be recognized 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 a `<newline>`).

Macro assignments shall 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 shall 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 makefile(s), processed in the order specified.

(4) Macros specified on the command line. It is unspecified whether the internal macros defined in 6.2.7.7 are accepted from the command line.

If the `−e` option is specified, the order of processing sources (2) and (3) shall be reversed.

The `SHELL` macro shall be treated specially. It shall be provided by `make` and set to the pathname of the shell command language interpreter (see `sh` in 4.56). The **SHELL** environment variable shall not affect the value of the `SHELL` macro. If `SHELL` is defined in the makefile or is specified on the command line, it shall replace the original value of the `SHELL` macro, but shall not affect the **SHELL** environment variable. Other effects of defining `SHELL` in the makefile or on the command line are implementation defined.

6.2.7.5 Inference Rules

Inference rules are formatted as follows:

target `:` 1
`<tab>`*command* 1
[`<tab>`*command*] 1
...

(*line that does not begin with* `<tab>` *or* #)

The *target* portion shall be a valid target name (see 6.2.7.3) and shall be of the 2
form *.s2* or *.s1.s2* (where *.s1* and *.s2* are suffixes that have been given as prere- 2
quisites of the `.SUFFIXES` special target and *s1* and *s2* do not contain any slashes 2
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 1
can have only one target before the colon. 1

The makefile shall not specify prerequisites for inference rules; no characters
other than white space shall follow the colon in the first line, except when creat- 1
ing the ''empty rule,'' described below. Prerequisites are inferred, as described 1
below. 1

Inference rules can be redefined. A target that matches an existing inference rule shall 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>s 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 shall be appended to the current list by specifying a `.SUFFIXES` special target in the makefile. A `.SUFFIXES` target with no prerequisites shall 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 6.2.7.7.)

When no target rule is found to update a target, the inference rules shall be 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 shall be 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 shall be 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 shall 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.

6.2.7.6 Libraries

If a target or prerequisite contains parentheses, it shall 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 shall 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 6.1. 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*.

6.2.7.7 Internal Macros

The `make` utility shall maintain 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) shall be updated. This shall be accomplished by recursively processing each prerequisite. Upon recursion, each prequisite shall become a target itself. Its prerequisites in turn shall be processed recursively until a target is found that has no prerequisites, at which point the recursion shall stop. The recursion then shall back 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, or

— An implicit prerequisite that becomes a target when `make` processes it during recursion.

In the definitions that follow, the word ''prerequisite'' refers to either:

— An explicit prerequisite specified in the makefile for a particular target, or

— 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:

`$@` The `$@` macro shall evaluate to the full target name of the current target, or the archive filename part of a library archive target. It shall be 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 shall be evaluated only when the current target is an archive library member of the form *libname* (*member* `.o`). In these cases, `$@` shall evaluate to *libname* and `$%` shall evaluate to *member* `.o`. The `$%` macro shall be evaluated for both target and inference rules.

For example, in a makefile target rule to build `lib.a(file.o)`, `$%` 1
represents `file.o`—as opposed to `$@`, which represents `lib.a`. 1

`$?` The `$?` macro shall evaluate to the list of prerequisites that are newer 1
than the current target. It shall be evaluated for both target and infer- 1
ence rules. 1

For example, in a makefile target rule to build `prog` from `file1.o`, 1
`file2.o`, and `file3.o`, and where `prog` is not out of date with respect 1
to `file1.o`, but is out of date with respect to `file2.o` and `file3.o`, 1
`$?` represents `file2.o` and `file3.o`. 1

`$<` In an inference rule, `$<` shall evaluate to the file name whose existence 1
allowed the inference rule to be chosen for the target. In the `.DEFAULT` 1
rule, the `$<` macro shall evaluate to the current target name. The `$<` 1
macro shall be evaluated only for inference rules. 1

For example, in the `.c.a` inference rule, `$<` represents the prerequisite 1
`.c` file. 1

`$*` The `$*` macro shall evaluate to the current target name with its suffix 1
deleted. It shall be evaluated at least for inference rules. 2

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 alternate form. When an uppercase `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`

6.2.7.8 Default Rules

The default rules for `make` shall achieve results that are the same as if the following were used. Implementations that do not support the C Language Development Utilities Option may omit `CC`, `CFLAGS`, `YACC`, `YFLAGS`, `LEX`, `LFLAGS`, `LDFLAGS`, and the `.c`, `.y`, and `.l` inference rules. Implementations that do not support the FORTRAN Language Development Utilities Option may omit `FC`, `FFLAGS`, and the `.f` inference rules. Implementations may provide additional macros and rules.

NOTE: In a future version of this standard, the default rules may be specified separately from the make clause, such as with the language-dependent development options.

SUFFIXES AND MACROS

```
.SUFFIXES: .o .c .y .l .a .sh .f                                          1

MAKE=make
AR=ar
ARFLAGS=-rv
YACC=yacc
YFLAGS=
LEX=lex
LFLAGS=
LDFLAGS=
CC=c89
CFLAGS=-O
FC=fort77
FFLAGS=-O 1
```

SINGLE SUFFIX RULES

```
.c:
      $(CC) $(CFLAGS) $(LDFLAGS) -o $@ $<

.f:
      $(FC) $(FFLAGS) $(LDFLAGS) -o $@ $<

.sh:
      cp $< $@
      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                                                       1
      mv y.tab.o $@

.l.o:
      $(LEX) $(LFLAGS) $<
      $(CC) $(CFLAGS) -c lex.yy.c
      rm -f lex.yy.c                                                      1
      mv lex.yy.o $@

.y.c:
      $(YACC) $(YFLAGS) $<
      mv y.tab.c $@

.l.c:
      $(LEX) $(LFLAGS) $<
      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
```

6.2.8 Exit Status

When the −q option is specified, the make utility shall 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 shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

6.2.9 Consequences of Errors

Default.

6.2.10 Rationale. *(This subclause is not a part of P1003.2)*

The make provided here is intended to provide the means for changing portable source code into runnable executables on a POSIX.2 system. It reflects the most common features present in System V and BSD makes.

Historically, the make utility has been an especially fertile ground for vendor- and research-organization-specific syntax modifications and extensions. Examples include:

— Syntax supporting parallel execution (Sequent, Cray, GNU, and others)

— Additional ‘‘operators’’ separating targets and their prerequisites (System V, BSD, and others)

— Specifying that command lines containing the strings ${MAKE} and $(MAKE) are executed when the −n option is specified (GNU and System V)

— Modifications of the meaning of internal macros when referencing libraries (BSD and others)

— Using a single instance of the shell for all of a target's command lines (BSD and others)

— Allowing spaces as well as tabs to delimit command lines (BSD)

— Adding C-preprocessor-style ''include'' and ''ifdef'' constructs (System V, GNU, BSD, and others)

— Remote execution of command lines (Sprite and others)

— Specifying additional special targets (Sun, BSD, System V, and most others).

Additionally, many vendors and research organizations have rethought the basic concepts of `make`, creating vastly extended, as well as completely new, syntaxes. Each of these versions of ''make'' fulfills the needs of a different community of users; it is unreasonable for this standard to require behavior that would be incompatible (and probably inferior) to existing practice for such a community.

In similar circumstances, when the industry has enough sufficiently incompatible formats as to make them irreconcilable, POSIX.2 has followed one or both of two courses of action. Commands have been renamed (`cksum`, `echo`, and `pax`) and/or command-line options have been provided to select the desired behavior (`grep`, `od`, and `pax`).

Because the syntax specified for the `make` utility is, by and large, a subset of the syntaxes accepted by almost all versions of `make`, it was decided that it would be counter-productive to change the name. And since the makefile itself is a basic unit of portability, it would not be completely effective to reserve a new option letter, such as `make -P`, to achieve the portable behavior. Therefore, the special target `.POSIX` was added to the makefile, allowing users to specify ''standard'' behavior. This special target does not preclude extensions in the `make` utility, or such extensions being used by the makefile specifying the target; it does, however, preclude any extensions from being applied that could alter the behavior of previously valid syntax; such extensions must be controlled via command-line options or new special targets. 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 portable version of `make` described in this clause is not intended to be the state of the art software generation tool and, as such, some newer and more leading-edge features have not been included. An attempt has been made to describe the portable makefile in a manner that does not preclude such extensions as long as they do not disturb the portable behavior described here.

One use of this `make` and the makefile syntax is as a format that newer versions of `make` can generate for portability purposes.

Examples, Usage

The following command:

```
make
```

makes the first target found in the makefile.

The following command:

```
make junk
```

makes the target `junk`.

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
```

An example for making optimized `.o` files from `.c` files is:

```
.c.o:
        c89 −c −O $*.c
```

or:

```
.c.o:
        c89 −c −O $<
```

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 1
into `lib`. 1

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 behavior 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* .

The definition of **MAKEFLAGS** allows both the System V letter string and the BSD command-line formats. The two formats are sufficiently different to allow implementations to support both without ambiguity.

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='#'"
```

Earlier drafts stated that an ‘‘unquoted’’ number sign was treated as the start of a comment. The `make` utility does not pay any attention to quotes. A number sign starts a comment regardless of its surroundings.

The treatment of escaped `<newline>`s 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
```

The contents of the built-in rules can be viewed by running:

```
make -p -f /dev/null 2>/dev/null
```

Many historical `make`s stop chaining together inference rules when an intermedi- 1
ate target is nonexistent. For example, it might be possible for a `make` to deter- 1
mine that both `.y.c` and `.c.o` could be used to convert a `.y` to a `.o`. Instead, in 1
this case, `make` requires the use of a `.y.o` rule. 1

The text about ‘‘other implementation-defined pathnames may also be tried’’ in addition to `./makefile` and `./Makefile` is to allow such extensions as `SCCS/s.Makefile` and other variations. It was made an implementation-defined requirement (as opposed to unspecified behavior) to highlight surprising implementations that might select something unexpected like `/etc/Makefile`.

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`’’.)

History of Decisions Made

Earlier drafts contained the macro NPROC as a means of specifying that make should use *n* processes to do the work required. While this feature is a valuable extension for many systems, it is not common usage and could require other nontrivial extensions to makefile syntax. This extension is not required by the standard, but could be provided as a compatible extension. The macro PARALLEL is used by some historical systems with essentially the same meaning (but without using a name that is a common system limit value). It is suggested that implementors recognize the existing use of NPROC and/or PARALLEL as extensions to make.

The default rules are based on System V. The default CC= value is c89 instead of cc because POSIX.2 does not standardize the utility named cc. Thus, every conforming application would be required to define CC=c89 to expect to run. There is no advantage conferred by the hope that the makefile might hit the ‘‘preferred’’ compiler because there is no way that this can be guaranteed to work. Also, since the portable makescript can only use the c89 options, no advantage is conferred in terms of what the script can do. It is a quality of implementation issue as to whether c89 is as good as cc.

Since SCCS and RCS are not part of POSIX.2, all make references to SCCS extensions have been omitted.

The −d option to make is frequently used to produce debugging information, but is too implementation-dependent to add to the standard.

The −p option is not passed in **MAKEFLAGS** on most existing implementations and to change this would cause many implementations to break without sufficiently increased portability.

Commands that begin with a plus-sign (+) are executed even if the −n option is present. Based on the GNU version of make, the behavior of −n when the plus-sign prefix is encountered has been extended to apply to −q and −t as well. However, the System V convention of forcing command execution with −n when a target's command line contains either of the strings $(MAKE) or ${MAKE} has not been adopted. This functionality appeared in earlier drafts, but the danger of this approach was pointed out with the following example of a portion of a makefile:

```
subdir:
        cd subdir; rm all_the_files; $(MAKE)
```

The loss of the System V behavior in this case is well-balanced by the safety afforded to other makefiles that were not aware of this situation. In any event, the command-line plus-sign prefix can provide the desired functionality.

The double colon in the target rule format is supported in BSD systems to allow more than one target line containing the same target name to have commands associated with it. Since this is not functionality described in the *SVID* or XPG3, it has been allowed as an extension, but not mandated.

The default rules are provided with text specifying that the built-in rules are to be the same *as if* the listed set were used. The intent is that implementations should be able to use the rules without change, but will be allowed to alter them in ways that do not affect the primary behavior.

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.

The argument could be made to drop the default rules list from the standard. They provide convenience, but do not enhance portability of applications. The prime benefit is in portability of users who wish to type `make command` and have the command build from a `command.c` file.

The historical **MAKESHELL** feature was omitted. In some implementations it is used to provide a way of letting a user override the shell to be used to run `make` commands. This was confusing; for a portable `make`, the shell should be chosen by the makefile writer or specified on the `make` command line and not by a user running `make`.

The `make` utilities in most historical implementations process the prerequisites of a target in left-to-right order, and the POSIX.2 makefile format requires this. It supports the standard idiom used in many makefiles that produce `yacc` programs, for example:

```
    foo:    y.tab.o lex.o main.o
            $(CC) $(CFLAGS) -o $@ t.tab.o lex.o main.o
```

In this example, if `make` chose any arbitrary order, the `lex.o` might not be made with the correct `y.tab.h`. Although there may be better ways to express this relationship, it is widely used historically. Implementations that desire to update prerequisites in parallel should require an explicit extension to `make` or the makefile format to accomplish it, as described previously.

The algorithm for determining a new entry for target rules is partially unspecified. Some historical `make`s allow blank, empty, or comment lines within the collection of commands marked by leading <tab>s. A conforming makefile must ensure that each command starts with a <tab>, but implementations are free to ignore blank, empty, and comment lines without triggering the start of a new entry.

The Asynchronous Events subclause includes having SIGTERM and SIGHUP, along with the more traditional SIGINT and SIGQUIT, remove the current target unless directed not to. SIGTERM and SIGHUP were added to parallel other utilities that have historically cleaned up their work as a result of these signals. All but SIGQUIT is required to resend itself the signal it received to cause `make` to exit with a status that reflects the signal. The results from SIGQUIT are partially unspecified because, on systems that create `core` files upon receipt of SIGQUIT, the `core` from `make` would conflict with a core file from the command that was running when the SIGQUIT arrived. The main concern here was to prevent damaged files from appearing up-to-date when `make` is rerun.

The `.PRECIOUS` special target was extended to globally affect all targets (by specifying no prerequisites). The `.IGNORE` and `.SILENT` special targets were extended to allow prerequisites; it was judged to be more useful in some cases to be able to turn off errors or echoing for a list of targets than for the entire makefile. These extensions to System V's `make` were made to match historical practice from the BSD `make`.

Macros are not exported to the environment of commands to be run. This was never the case in any historical `make` and would have serious consequences. The environment is the same as the environment to `make` except that **MAKEFLAGS** and macros defined on the `make` command line are added.

Some implementations do not use *system* () for all command lines, as required by the POSIX.2 portable makefile format; as a performance enhancement, they select lines without shell metacharacters for direct execution by *execve* (). There is no requirement that *system* () be used specifically, but merely that the same results be achieved. The metacharacters typically used to bypass the direct *execve* () execution have been any of:

```
= | ^ ( ) ; & < > * ? [ ] : $ ‘ ’ " \ \n
```

The default in some advanced versions of `make` is to group all the command lines for a target and execute them using a single shell invocation; the System V method is to pass each line individually to a separate shell. The single-shell method has the advantages in performance and the lack of a requirement for many continued lines. However, converting to this newer method has caused portability problems with many historical makefiles, so the behavior with the POSIX makefile is specified to be the same as System V's. It is suggested that the special target `.ONESHELL` be used as an implementation extension to achieve the single-shell grouping for a target or group of targets.

Novice users of `make` have had difficulty with the historical need to start commands with a `<tab>` character. Since it is often difficult to discern differences between `<tab>` and `<space>` characters on terminals or printed listings, confusing bugs can arise. In earlier drafts, an attempt was made to correct this problem by allowing leading `<blank>`s instead of `<tab>`s. However, implementors reported many makefiles that failed in subtle ways following this change and it is difficult to implement a `make` that unambiguously can differentiate between macro and command lines. There is extensive historical practice of allowing leading spaces before macro definitions. Forcing macro lines into column 1 would be a significant backward compatibility problem for some makefiles. Therefore, historical practice was restored.

The System V INCLUDE feature was considered, but not included. This would treat a line that began in the first column and contained INCLUDE *<filename>* as an indication to read *<filename>* at that point in the makefile. This is difficult to use in a portable way and it raises concerns about nesting levels and diagnostics. System V, BSD, GNU, and others have used different methods for including files.

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.

The System V dynamic dependency feature was not added. It would support:

```
cat: $$@.c
```

that would expand to

```
cat: cat.c
```

This feature exists only in the new version of System V `make` and, while useful, is not in wide usage. This means that macros are expanded twice for prerequisites: once at makefile parse time and once at target update time.

Consideration was given to adding metarules to the POSIX `make`. This would make `"%.o: %.c"` the same as `".c.o:"`. This is quite useful and available from some vendors, but it would cause too many changes to this `make` to support. It
would have introduced rule chaining and new substitution rules. However, the 1
rules for target names have been set to reserve the `%` and `"` characters. These are 1
traditionally used to implement metarules and quoting of target names, respec- 1
tively. Implementors are strongly encouraged to use these characters only for 1
these purposes. 1

A request was made to extend the suffix delimiter character from a period to any character. The metarules in newer `make`s solves this problem in a more general way. POSIX.2 is staying with the more conservative historical definition until a clear industry consensus on `make` technology might prompt a revision of this standard.

The standard output format for the `-p` option is not described because it is primarily a debugging option and the format is not generally useful to programs. In historical implementations the output is not suitable for use in generating makefiles. The `-p` format has been variable across historical implementations. Therefore, the definition of `-p` was only to provide a consistently named option for obtaining `make` script debugging information.

Some historical implementations have not cleared the suffix list with `-r`.

Implementations should be aware that some historical applications have intermixed *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, but some backward compatibility support may be warranted.

Empty inference rules are specified with a semicolon command rather than omitting all commands, as described in a previous draft. The latter case has no traditional meaning and is reserved for implementation extensions, such as in GNU `make`.

6.3 `strip` — Remove unnecessary information from executable files

6.3.1 Synopsis

`strip` *file* . . .

6.3.2 Description

The `strip` utility shall remove from executable files named by the *file* operands any information the implementor deems unnecessary to proper execution of those files. The nature of that information is unspecified. The effect of `strip` shall be the same as the use of the `−s` option to any of the compilers defined by this standard.

6.3.3 Options

None.

6.3.4 Operands

The following operand shall be supported by the implementation:

file A pathname referring to an executable file.

6.3.5 External Influences

6.3.5.1 Standard Input

None.

6.3.5.2 Input Files

The input files shall be in the form of executable files successfully produced by any compiler defined by this standard.

6.3.5.3 Environment Variables

The following environment variables shall affect the execution of `strip`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments).

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

6.3.5.4 Asynchronous Events

Default.

6.3.6 External Effects

6.3.6.1 Standard Output

None.

6.3.6.2 Standard Error

Used only for diagnostic messages.

6.3.6.3 Output Files

The `strip` utility shall produce executable files of unspecified format.

6.3.7 Extended Description

None.

6.3.8 Exit Status

The `strip` utility shall exit with one of the following values:

0 Successful completion.

>0 An error occurred.

6.3.9 Consequences of Errors

Default.

6.3.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

None.

History of Decisions Made

Historically, this utility has been used to remove the symbol table from an executable file. It was included since it is known that the amount of symbolic information can amount to several megabytes; the ability to remove it in a portable manner was deemed important, especially for smaller systems.

The behavior of `strip` is said to be the same as the `−s` option to a compiler. While the end result is essentially the same it is not required to be identical. The same effect can be achieved with either `−s` during a compile or a `strip` on the final object file.

Section 7: Language-Independent System Services

This clause contains functional specifications for services that give applications access to features defined elsewhere in this standard. These services allow applications written in high-level languages to

(1) execute commands using the shell language,

(2) obtain values of environment variables,

(3) perform regular expression and pattern matching,

(4) process command arguments in a standard manner,

(5) generate pathnames from a pattern,

(6) perform shell word expansions,

(7) obtain system configuration information, and

(8) set locale control information

This clause does not define interfaces, but services that shall be provided by the interfaces in a language-dependent binding. This clause is optional, in that an implementation is not required to support any language binding to these services. However, any language binding shall support all of the services described here. Implementations therefore provide support for services in this clause by supplying a language-dependent binding such as the one defined in Annex B. Such a system would specify conformance to the language-dependent binding, not to the language-independent bindings given here.

7.0.1 Language-Independent System Services Rationale. *(This subclause is not a part of P1003.2)*

Section 7 essentially is a metastandard, in that it specifies services that must be in a language-dependent binding. An implementation conforms to a specific language-dependent binding such as for the C language, in Annex B, and the language-dependent binding must conform to the specifications in this clause.

In this standard, the language-independent specifications have not yet been developed. The language-independent syntax is being created in parallel by the POSIX. 1 working group. Therefore, the C language bindings temporarily described in Annex B are actually the full interface specifications. It is the intention of the P1003.2 working group to rectify this situation in a later supplement by moving the majority of the interface specifications back into this clause, leaving Annex B with only brief descriptions of the C bindings to those services.

This clause does not attempt to include everything that would be required of a language binding. The services here are those that are necessary to make use of features defined elsewhere in the standard, but that are not normally available in every language. Clearly a language that could not open, read, and write the files manipulated by the utilities in this standard would not be very useful, but this service is normally provided by any language and therefore isn't called out here. The ability to obtain values of environment variables exported from the shell, on the other hand, is not universally available, so that service is included here.

7.1 Shell Command Interface

7.1.1 Execute Shell Command

Any language binding to Language-Independent System Services shall include a facility to execute a shell command.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *system* () function described in B.3.1.

7.1.2 Pipe Communications with Programs

Any language binding to Language-Independent System Services shall include a facility to execute a shell command, and to write the standard input or read the standard output of that command via a pipe.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *popen* () and *pclose* () functions described in B.3.2.

7.2 Access Environment Variables

Any language binding to Language-Independent System Services shall include a facility to obtain values of environment variables, as specified in POSIX.1 {8}.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *getenv* () function described in POSIX.1 {8} 4.6.1.

7.2.1 Access Environment Variables Rationale. *(This subclause is not a part of P1003.2)*

This facility is required in POSIX.2 so that applications can obtain values of exported shell variables.

7.3 Regular Expression Matching

Any language binding to Language-Independent System Services shall include a facility to interpret regular expressions as described in 2.8.

The language-independent specification for this facility has not been developed. The C binding is the *regcomp* (), *regexec* (), and *regfree* () functions described in B.5.

7.3.1 Regular Expression Matching Rationale. *(This subclause is not a part of P1003.2)*

This service is important enough that it should be required by any language binding to POSIX.2.

Regular expression parsing and pattern matching are listed separately, since they are different services. A language binding could provide different functions to support regular expressions and patterns, or could combine them into a single function.

7.4 Pattern Matching

Any language binding to Language-Independent System Services shall include a facility to interpret patterns as described in 3.13.1 and 3.13.2. This facility shall allow the application to specify whether a slash character in the string to be matched will be treated as a regular character, or must be explicitly matched against a slash in the pattern.

The language-independent specification for this facility has not been developed. The C binding is the *fnmatch* () function described in B.6.

7.5 Command Option Parsing

Any language binding to Language-Independent System Services shall include a facility to parse the options and operands from the command line that invoked the application.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *getopt* () function described in B.7.

7.6 Generate Pathnames Matching a Pattern

Any language binding to Language-Independent System Services shall include a facility to generate pathnames matching a pattern as described in 3.13.

The language-independent specifications for this facility has not been developed. The C binding is the *glob* () and *globfree* () functions described in B.8.

7.7 Perform Word Expansions

Any language binding to Language-Independent System Services shall include a facility to do shell word expansions as described in 3.6.

The language-independent specification for this facility has not been developed. The C binding is the *wordexp* () and *wordfree* () functions described in B.9.

7.7.1 Perform Word Expansions Rationale. *(This subclause is not a part of P1003.2)*

See the rationale for this function in B.9.

7.8 Get POSIX Configurable Variables

7.8.1 Get String-Valued Configurable Variables

Any language binding to Language-Independent System Services shall include a facility to obtain string configurable variables.

The language-independent specification for this facility has not been developed. The C binding for this facility is the *confstr* () function described in B.10.1.

7.8.2 Get Numeric-Valued Configurable Variables

Any language binding to Language-Independent System Services shall include facilities to determine the current values of system and pathname limits or options (*variables*), as specified by POSIX.1 {8}. The configurable variables listed in Table 7-1, which are defined in POSIX.1 {8}, shall be available in any POSIX.2 language-dependent binding, with minimum values as given in POSIX.1 {8}. Other POSIX.1 {8} configurable variables may be supported, but are not required by POSIX.2. This facility shall also make available current values for all system limits defined in 2.13.

The language-independent specifications for these facilities have not been developed. The C bindings are the *sysconf* () function described in POSIX.1 {8}4.8, and the *pathconf* () and *fpathconf* () functions defined in POSIX.1 {8}5.7.

7.8.2.1 Get Numeric-Valued Configurable Variables Rationale. *(This subclause is not a part of P1003.2)*

This description calls out specific values that *sysconf* (), *pathconf* (), and *fpathconf* () are required to support. Some of the POSIX.1 {8} values are excluded from this list because they are not relevant in a POSIX.2-only environment. Currently, only {CLK_TCK} is not required by POSIX.2.

This description does not specify the *name* values for the arguments to the various functions. This is because different language bindings might use different naming conventions, or might use a completely different scheme for obtaining the required configurable values. Specific names for the *name* values for the C language binding are given in B.10.2.

7.9 Locale Control

Any language binding to Language-Independent System Services shall include a facility to set locale control information.

The language-independent specification for this facility has not been developed. The C binding for this facility is described in B.11.

7.9.0.1 Locale Control Rationale. *(This subclause is not a part of P1003.2)*

This facility is required in POSIX.2 so that applications can control the locale, which affects the operation of POSIX.2 utilities.

Table 7-1 – POSIX.1 Numeric-Valued Configurable Variables

{ARG_MAX}	{NAME_MAX}	{_POSIX_CHOWN_RESTRICTED}
{CHILD_MAX}	{NGROUPS_MAX}	{_POSIX_JOB_CONTROL}
{LINK_MAX}	{OPEN_MAX}	{_POSIX_NO_TRUNC}
{MAX_CANON}	{PATH_MAX}	{_POSIX_SAVED_IDS}
{MAX_INPUT}	{PIPE_BUF}	{_POSIX_VDISABLE}

Annex A
(normative)

C Language Development Utilities Option

This annex describes utilities used for the development of C language applications, including compilation or translation of C source code and complex program generators for simple lexical tasks and processing of context-free grammars.

The utilities described in this annex may be provided by the conforming system; however, any system claiming conformance to the **C Language Development Utilities Option** shall provide all of the utilities described here. The utilities described in Section 6 are prerequisites to this annex.

A.0.1 C Language Development Utilities Option Rationale. *(This subclause is not a part of P1003.2)*

The portions of this standard that concern specific languages—currently C and FORTRAN—have been collected to the rear of the document as Normative Annexes. For purposes of conformance, they are no less a part of the standard than one of the numbered sections. They were grouped as Annexes to illustrate that the base standard is [planned to be] language independent, giving a small degree of separation. The working group also wished to send a message to those groups planning other language bindings: the standard is not C-oriented, and there's plenty of room to add more annexes for your languages as you develop them, right alongside C and FORTRAN.

A.1 c89 — Compile Standard C programs

A.1.1 Synopsis

`c89` [`−c`] [`−D` *name* [`=`*value*]] . . . [`−E`] [`−g`] [`−I` *directory*] . . . [`−L` *directory*] . . .
[`−o` *outfile*] [`−O`] [`−s`] [`−U` *name*] . . . *operand* . . .

A.1.2 Description

The `c89` utility is the interface to the standard C compilation system; it shall accept source code conforming to the C Standard {7}. The system conceptually consists of a compiler and link editor. The files referenced by *operand* s shall 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`

shall 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 shall be written according to the `−o` *outfile* option (if present) or to the file `a.out`.

The executable file shall be created as specified in 2.9.1.4, except that the file permissions shall be set to

S_IRWXO | S_IRWXG | S_IRWXU

(see 5.6.1.2 in POSIX.1 {8}) and that the bits specified by the *umask* of the process shall be cleared.

A.1.3 Options

The `c89` utility shall conform to the utility argument syntax guidelines described in 2.10.2, 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. 1

— Conforming applications shall specify each option separately; that is, grouping option letters (e.g., `−cO`) need not be recognized by all implementations.

The following options shall be supported by the implementation:

`−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-defined interactions with other options.

`−s` Produce object and/or executable files from which symbolic and other information not required for proper execution using *exec* (see POSIX.1 {8} 3.1.2) 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 shall 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* shall be undefined regardless of the order of the options. Additional implementation-defined *names* may be provided by the compiler. Implementations shall support at least 2048 bytes of `−D` definitions and 256 *names*.

`−E` Copy C-language source files to the standard output, expanding all preprocessor directives; no compilation shall 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 (`""`) shall 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 shall be searched for only in directories named in `−I` options and then in the usual places. Directories named in `−I` options shall be searched in the order specified. Implementations shall 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 shall be searched in the order specified. Implementations shall 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` Optimize. The nature of the optimization is unspecified.

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

Multiple instances of the `−D`, `−I`, `−U`, and `−L` options can be specified.

A.1.4 Operands

An *operand* is either in the form of a pathname or the form `−l` *library* . At least one operand of the pathname form shall be specified. The following operands shall be supported by the implementation:

file.`c` A C-language source file to be compiled and optionally linked. The operand shall be of this form if the `−c` option is used.

file.`a` A library of object files typically produced by `ar` (see 6.1), and passed directly to the link editor. Implementations may recognize implementation-defined 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 recognize implementation-defined suffixes other than `.o` as denoting object files.

The processing of other files is implementation defined.

`−l` *library* (The letter ell.) Search the library named:

`lib`*library* `.a`

A library shall 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 A.1.7. Implementations may recognize implementation-defined suffixes other than `.a` as denoting libraries.

A.1.5 External Influences

A.1.5.1 Standard Input

None.

A.1.5.2 Input Files

The input file shall 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 defined.

A.1.5.3 Environment Variables

The following environment variables shall affect the execution of `c89`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

TMPDIR This variable shall be interpreted as a pathname that should override the default directory for temporary files, if any.

A.1.5.4 Asynchronous Events

Default.

A.1.6 External Effects

A.1.6.1 Standard Output

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, shall precede the processing of each input file; they shall not be written to standard output if they are written to standard error, as described in A.1.6.2.

If the `-E` option is specified, the standard output shall be a text file that 1
represents the results of the preprocessing stage of the language; it may contain 1
extra information appropriate for subsequent compilation passes. 1

A.1.6.2 Standard Error

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, shall precede the processing of each input file; they shall not be written to the standard error if they are written to the standard output, as described in A.1.6.1.

This utility may produce warning messages about certain conditions that do not warrant returning an error (nonzero) exit value.

A.1.6.3 Output Files

Object files or executable files or both are produced in unspecified formats.

A.1.7 Extended Description

A.1.7.1 Standard Libraries

The `c89` utility shall recognize the following `−l` operands for standard libraries:

`−l c` This library contains all library functions referenced in `<stdlib.h>`, `<stdio.h>`, `<time.h>`, `<setjmp.h>`, `<signal.h>`, `<unistd.h>`, `<sys/types.h>`, `<string.h>`, and `<ctype.h>`, except for those functions referenced in `<math.h>`. If an invocation of

```
getconf _POSIX_VERSION
```

exits with a status of zero, the library searched also shall include all functions defined by POSIX.1 {8}; if the status is nonzero, it is unspecified whether these functions are available. If an invocation of

```
getconf _POSIX2_C_BIND
```

exits with a status of zero, the library searched also shall include all functions specified in Annex B; if the status is nonzero, it is unspecified whether these functions are available. An implementation shall not require this operand to be present to cause a search of this library.

`−l m` This library contains 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` (see A.2) that are not made available through the `−l c` operand.

`−l y` This library contains all functions required by the C-language output of `yacc` (see A.3) 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 shall 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`, and `liby.a` exist as regular files. The implementation may accept as `−l` operands names of objects that do not exist as regular files.

A.1.7.2 External Symbols

The C compiler and link editor shall support the significance of external symbols 1
up to a length of at least 31 bytes; the action taken upon encountering symbols exceeding the implementation-defined maximum symbol length is unspecified.

The compiler and link editor shall support a minimum of 511 external symbols per source or object file, and a minimum of 4095 external symbols total. A diagnostic message shall be written to the standard output if the implementation-defined limit is exceeded; other actions are unspecified.

A.1.8 Exit Status

The `c89` utility shall exit with one of the following values:

0 Successful compilation or link edit.

>0 An error occurred.

A.1.9 Consequences of Errors

When `c89` encounters a compilation error that causes an object file not to be created, it shall write a diagnostic to standard error and continue to compile other source code operands, but it shall not perform the link phase and shall return a nonzero exit status. If the link edit is unsuccessful, a diagnostic message shall be written to standard error and `c89` shall exit with a nonzero status.

A.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Note that some implementations support a finer-grained model of compilation than the one described above. In this model, the following conceptual phases may exist: preprocessor, compiler, optimizer, assembler, link editor. Such implementations may support these additional options to the `c89` utility:

−P Preprocess, but do not compile, the named C programs and leave the result on corresponding files suffixed .i.

−S Compile the named C programs into assembly language, and leave the assembler-language output on corresponding files suffixed .s. No object files are created.

[−W*c*,*arg1* [,*arg2* . . .]]
Hand off the argument(s) *argi* to phase *c* where *c* is one of [p02al] indicating preprocessor, compiler, optimizer, assembler, or link editor, respectively. For example, −Wa,−m passes −m to the assembler phase. (Note the rationale concerning −W in 2.10.1.1.)

The −fpq options have been excluded, since they use features that are not in this standard.

In specifying that *file*.a operands are *typically* produced by ar, it is the intention of POSIX.2 to require that object libraries produced by ar be usable by c89, but not to preclude an implementation from supplying another utility that creates object library files.

The following are examples of usage:

`c89 -o foo foo.c`	Compiles foo.c and creates the executable 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 a.out.
`c89 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:

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.

Multiple −L operands can be used when library name collisions occur. Building on the previous example, suppose that we now want to use a new libp.a, in /a/a/a, but we still want *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.

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 namespace, etc.), so the existence of files named in the same manner as the standard libraries within the `−L` directories is explicitly stated to produce unspecified behavior.

Some historical implementations have permitted `−L` options to be interspersed with `−l` operands on the command line; with respect to POSIX, such behavior would be considered a vendor extension. For an application to compile consistently on systems that do not behave like this, it is necessary for a conforming application to supply all `−L` options before any of the `−l` options.

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.

History of Decisions Made

The name of this utility differs from the historical `cc` name. The C Standard {7} document was approved during the development of POSIX.2, and it is clear that POSIX must support Standard C; there is no other good way of specifying a C language. The support of the C Standard {7} by `c89` also mandates the Standard C math libraries. An alternative approach was considered: provide an option to select the type of compilation required. However, it was found that all available option letters were already in use in the various historical `cc` utilities. Thus, this name change is being used essentially as a switch. There was some temptation to use the name change as an excuse to mandate a cleaner interface (e.g., conform to the utility syntax guidelines), but this was resisted; the majority of early `c89` implementations are expected to be satisfied with historical `cc`s with only minimal changes. This was decided more from the standpoint of existing applications and makefiles than for the implementors' sake.

The `−l` *library* operand must be capable of being interspersed with file name operands so that the order in which libraries are searched by the link editor can be specified.

The search algorithm for `−I` *directory* states that the directory of the file with the `#include` file is searched first, rather than being implementation defined. It is believed that this reflects most implementations, and it disallows variations on different implementations, since this would make it very difficult to distribute source code in a compatible form.

The `−I` options are searched in the order specified (which is left to right in English). This resolves the conflict of what header file is used if multiple files with the same name exist in different directories in the `include` path.

In a future extension or supplement to this standard, *should* will be changed to *shall* with respect to support for **TMPDIR** by applications.

It is unclear whether `c89` requires such a large number of file descriptors that its requirement should be documented here; POSIX.2 remains silent on the issue. It is also noted that an undocumented feature of some C compilers is that if file descriptor 9 is open, a linkage trace is written to it.

There is no pseudo-*printf*() specification for compile errors because no common format could be identified. As new C compilers are written, they are encouraged to use the following format:

> `"%s: %s: %d %s\n"`, *<compiler phase>*, *<file name>*, *<line number>*, *<explanation>*

The following option proposals were considered and rejected:

(1) The `−M` option in BSD does not exist in System V, and is not seen to enhance application portability.

(2) The `−S` option was not seen to enhance application portability, and makes assumptions about the underlying architecture.

Earlier drafts included a `−V` option to select a compiler version. Not only did this letter (and every other upper- and lowercase letter) collide with one historical implementation or another, but there was no agreement on how many compiler versions should be defined, or what they should mean. Another choice is to specify that the `cc` utility invoke a Standard C compiler. By specifying `c89` instead, an installation is able to link either a ‘‘common usage’’ or a Standard C compiler to the name `cc`. Implementors are free to select implementation-defined options to select (nonportable) extensions to their existing C compiler to aid the transition to Standard C.

The `−g` and `−s` options are not specified as mutually exclusive. Historically these two options have been mutually exclusive, but because both are so loosely specified, it seemed cleaner to leave their interaction unspecified.

The `−E` option was added because headers are not required to be separate files in a POSIX.1-conformant system; these values could be hard-coded into the compiler, or might only be accessible in a nonportable way. Hence, while not strictly required for application portability, this option is a practical necessity as a portable means for ascertaining the real effects of preprocessor statements.

In BSD systems, using `−c` and `−o` in the same command causes the object module to be stored in the specified file. In System V, this produces an error condition. Therefore, POSIX.2 indicates that this is an unspecified condition.

Reasonably precise specification of standard library access is required. Implementations are not required to have `/usr/lib/libc.a`, etc., as many historical implementations do, but if not they are required to recognize `c`, `m`, `l`, and `y` as tokens. Libraries `l` and `y` can be empty if the library functions specified for `lex` and `yacc` are accessible through the `−l c` operand. Historically, these libraries have been necessary, but they are not required for a conforming implementation.

External symbol size limits are in a normative subclause; portable applications need to know these limits. However, the minimum maximum symbol length should be taken as a constraint on a portable application, not on an implementation, and consequently the action taken for a symbol exceeding the limit is unspecified. The minimum size for the external symbol table was added for similar reasons.

The Consequences of Errors subclause clearly specifies the compiler's behavior when compilation or link-edit error occur. The behavior of several historical implementations was examined, and the choice was made to be silent on the status of the executable, or `a.out`, file in the face of compiler or linker errors. If a linker writes the executable file, then links it on disk with *lseek* ()s and *write* ()s, the partially-linked executable can be left on disk and its execute bits turned off if the link edit fails. However, if the linker links the image in memory before writing the file to disk, it need not touch the executable file (if it already exists) because the link edit fails. Since both approaches are existing practice, a portable application shall rely on the exit status of `c89`, rather than on the existence or mode of the executable file.

The requirement that portable applications specify compiler options separately is to reserve the multicharacter option namespace for vendor-specific compiler options, which are known to exist in many historical implementations. Implementations are not required to recognize, for example `−gc` as if it were `−g −c`; nor are they forbidden from doing so. The synopsis shows all of the options separately to highlight this requirement on applications.

Echoing filenames to standard error is considered a diagnostic message, because it might otherwise be difficult to associate an error message with the erring file. The text specifies either standard error or standard output for these messages because some historical practice uses standard output, but there was considerable sentiment expressed for allowing it to be on standard error instead. The rationale for using standard output is that these are not really error message headers, but a running progress report on which files have been processed. The messages are described as optional because there might be different ways of constructing the compiler's messages that should not be precluded.

A.2 `lex` — Generate programs for lexical tasks

A.2.1 Synopsis

`lex` [−t] [−n | −v] [*file* . . .]

Obsolescent Version :

`lex` −c [−t] [−n | −v] [*file* . . .]

A.2.2 Description

The `lex` utility shall generate C programs to be used in lexical processing of character input, and that can be used as an interface to `yacc` (see A.3). The C programs shall be generated from `lex` source code and conform to the C Standard {7}. 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 nonzero exit status. See A.2.7 for a complete description of the `lex` input language.

A.2.3 Options

The `lex` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

−c (Obsolescent.) 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 A.2.7.1.) If the −t option is specified and −n is not specified, this report shall 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.

A.2.4 Operands

The following operand shall be supported by the implementation:

file A pathname of an input file. If more than one such *file* is specified, all files shall be concatenated to produce a single `lex` program. If no *file* operands are specified, or if a *file* operand is −, the standard input shall be used.

A.2.5 External Influences

A.2.5.1 Standard Input

The standard input shall be used if no *file* operands are specified, or if a *file* operand is −. See Input Files.

A.2.5.2 Input Files

The input files shall be text files containing `lex` source code, as described in A.2.7.

A.2.5.3 Environment Variables

The following environment variables shall affect the execution of `lex`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_COLLATE This variable shall determine the locale for the behavior of ranges, equivalence classes, and multicharacter collating elements within regular expressions. If this variable is not set to the POSIX Locale, the results are unspecified.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files) and the behavior of character classes within extended regular expressions. If this variable is not set to the POSIX Locale, the results are unspecified.

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

A.2.5.4 Asynchronous Events

Default.

A.2.6 External Effects

A.2.6.1 Standard Output

If the `-t` option is specified, the text file of C source code output of `lex` shall be written to standard output.

If the `-t` option is not specified:

(1) Implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall 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 shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a `%` operator in the *Definitions* section (see A.2.7), as long as the `-n` option is not specified.

A.2.6.2 Standard Error

If the `-t` option is specified, implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall be written to the standard error.

If the `-t` option is not specified:

(1) Implementation-defined informational, error, and warning messages concerning the contents of `lex` source code input shall 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 shall also be written to either the standard output or standard error, in an implementation-defined format. These statistics may also be generated if table sizes are specified with a `%` operator in the *Definitions* section (see A.2.7), as long as the `-n` option is not specified.

A.2.6.3 Output Files

A text file containing C source code shall be written to `lex.yy.c`, or to the standard output if the `-t` option is present.

A.2.7 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`), 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 shall 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 A.2.7.1, the type can be explicitly selected using the `%array` or `%pointer` declarations, but the default is implementation defined.

— 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` shall search the set of patterns for the single long- 1
est possible match. Among rules that match the same number of characters, the 1
rule given first shall 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>` shall be assumed to be a C program fragment and shall 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 `%}` shall also be copied unchanged to the external definition area of the `lex.yy.c` file.

Any such input (beginning with a `<blank>` or within `%{` and `%}` delimiter lines) appearing at the beginning of the *Rules* section before any rules are specified shall 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>` 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.

A.2.7.1 `lex` Definitions

Definitions appear before the first `%%` delimiter. Any line in this section not contained between `%{` and `%}` lines and not beginning with a `<blank>` shall be 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 C Standard {7}, the result is undefined. The string *substitute* shall replace the string {*name* } when it is used in a rule. The *name* string shall be recognized 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` shall define a set of start conditions. Any line beginning with a `%` followed by a word beginning with either `x` or `X` shall define a set of exclusive start conditions. When the generated scanner is in a `%s` state, patterns with no state specified shall be also active; in a `%x` state, such patterns shall not be active. The rest of the line, after the first word, shall be considered to be one or more `<blank>`-separated names of start conditions. Start condition names shall be 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 the section A.2.7.4.

Implementations shall 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 defined. If an application refers to *yytext* outside of the scanner source file (i.e., via an `extern`), the application shall include the appropriate `%array` or `%pointer` declaration in the scanner source file.

Implementations shall accept declarations in the *Definitions* section for setting certain internal table sizes. The declarations are shown in Table A-1. In the table, *n* represents a positive decimal integer, preceded by one or more `<blank>`s. The exact meaning of these table size numbers is implementation defined. The implementation shall 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 shall be 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 shall provide.

Table A-1 – lex Table Size Declarations

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

A.2.7.2 `lex` Rules

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 recognized.

ERE action
ERE action
...

The extended regular expression (*ERE*) portion of a rule shall be separated from *action* by one or more `<blank>`s. A regular expression containing `<blank>`s shall be recognized under the following conditions: the entire expression appears within double-quotes; or, the `<blank>`s appear within double-quotes or square brackets; or, each `<blank>` is preceded by a backslash character.

A.2.7.3 `lex` User Subroutines

Anything in the user subroutines section shall be copied to `lex.yy.c` following 1
yylex (). 1

A.2.7.4 `lex` Regular Expressions

The `lex` utility shall support the set of extended regular expressions (see 2.8.4), with the following additions and exceptions to the syntax:

`"..."` Any string enclosed in double-quotes shall represent the charac- 1
ters within the double-quotes as themselves, except that 1
backslash escapes (which appear in Table A-2) shall be recog- 1
nized. Any backslash-escape sequence shall be terminated by the 1
closing quote. For example, `"\01""1"` represents a single string: 1
the octal value 1 followed by the character 1. 1

<*state*>*r* 1
<*state1* , *state2* , . . . >*r* 1
The regular expression *r* shall be matched only when the program 1

is in one of the start conditions indicated by *state* , *state1* , etc.; see 1
A.2.7.5. (As an exception to the typographical conventions of the
rest of this standard, in this case <*state*> does not represent a
metavariable, but the literal angle-bracket characters surround-
ing a symbol.) The start condition shall be recognized as such 1
only at the beginning of a regular expression. 1

r/*x* The regular expression *r* shall be matched only if it is followed by an occurrence of regular expression *x*. The token returned in *yytext* shall 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 (see A.2.7.1), the string, including the enclosing braces, shall be replaced by the *substitute* value. The *substitute* value shall be treated in the extended regular expression as if it were enclosed in parentheses. No substitution shall occur if `{`*name*`}` occurs within a bracket expression or within double-quotes.

Within an ERE, a backslash character shall be considered to begin an escape sequence as specified in Table 2-15 (see 2.12). In addition, the escape sequences in Table A-2 shall be recognized.

A literal `<newline>` character cannot occur within an ERE; the escape sequence 1
`\n` can be used to represent a `<newline>`. A `<newline>` shall not be matched by 2
a period operator. 2

The order of precedence given to extended regular expressions for `lex` differs from 2
that specified in Table 2-13. The order of precedence for `lex` shall be as shown in
Table A-3, from high to low.

NOTE: The escaped characters entry is not meant to imply that these are operators, but they are 2
included in the table to show their relationships to the true operators. The start condition, trailing 2
context, and anchoring notations have been omitted from the table because of the placement restric- 2
tions described in this subclause; they can only appear at the beginning or ending of an ERE. 2

The ERE anchoring operators (`^` and `$`) do not appear in Table A-3. With `lex` reg- 2
ular expressions, these operators are restricted in their use: the `^` operator can 2
only be used at the beginning of an entire regular expression, and the `$` operator 2
only at the end. The operators apply to the entire regular expression. Thus, for 2
example, the pattern `(^abc)|(def$)` is undefined; it can instead be written as 2
two separate rules, one with the regular expression `^abc` and one with `def$`, 2
which share a common action via the special `|` action (see below). If the pattern 2
were written `^abc|def$`, it would match either of `abc` or `def` on a line by itself. 2
Note also that `$` is a form of trailing context (it is equivalent to `/\n`) and as such 2

Table A-2 – `lex` Escape Sequences

Escape Sequence	Description	Meaning
digits	`<backslash>` followed by the longest sequence of one, two, or three octal-digit characters (01234567). If all of the digits are 0, (i.e., representation of the NUL character), the behavior 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-defined. Multibyte characters require multiple, concatenated escape sequences of this type, including the leading \ for each byte.
\x*digits*	`<backslash>` followed by the longest sequence of hexadecimal-digit characters (01234567abcdef ABCDEF). If all of the digits are 0, (i.e., representation of the NUL character), the behavior is undefined.	The character whose encoding is represented by the hexadecimal integer.
c	`<backslash>` followed by any character not described in this table or in Table 2-15	The character *c*, unchanged.

Table A-3 – `lex` ERE Precedence

collation-related bracket symbols	`[= =]` `[: :]` `[. .]`
escaped characters	\\<*special character* >
bracket expression	`[ ]`
quoting	`"..."`
grouping	`( )`
definition	{*name* }
single-character RE duplication	`* + ?`
concatenation	
interval expression	{*m*,*n*}
alternation	|

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 shall be special only in a start condition at the beginning of a regular

expression; elsewhere in the regular expression they shall be treated as ordinary 1
characters. 1

A.2.7.5 `lex` Actions

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 `;` shall be 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 shall not be valid, and the action `lex` takes in such a condition is undefined.

The specification for an action, including C statements and/or special actions, can extend across several lines if enclosed in braces:

ERE *<blank(s)>* { *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 shall be to copy the string to the output. Because the default behavior of a program generated by `lex` is to read the input and copy it to the output, a minimal `lex` source program that has just `%%` shall generate a C program that simply copies the input to the output unchanged.

Four special actions shall be available: ‘`|`’’,‘`ECHO;`’’,‘`REJECT;`’’,and ‘`BEGIN`’’: 1

`|` 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 shall be specified alone, with no other actions.

`ECHO;` Write the contents of the string *yytext* on the output. 1

`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. 1

`BEGIN` The

`BEGIN` *newstate* ;

action 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` (the `lex` library).

`int yylex(void)` Performs lexical analysis on the input; this is the primary function generated by the `lex` utility. The function shall return zero when the end of input is reached; otherwise it shall return nonzero values (tokens) determined by the actions that are selected.

`int yymore(void)` When called, indicates that when the next input string is recognized, it is to be appended to the current value of *yytext* rather than replacing it; the value in *yyleng* shall be 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* shall be adjusted accordingly.

`int input(void)` Returns the next character from the input, or zero on end of file. It shall obtain 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 *unput* ting 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 shall 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 shall 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.

Except for *input* (), *unput* (), and *main* (), all external and static names generated by `lex` shall begin with the prefix `yy` or `YY`.

A.2.8 Exit Status

The `lex` utility shall exit with one of the following values:

0	Successful completion.
>0	An error occurred.

A.2.9 Consequences of Errors

Default.

A.2.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

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("Unrecognized 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();
}
```

The following examples have been included to 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.

Unlike the general ERE rules, embedded anchoring is not allowed by most histori- 2
cal `lex` implementations. An example of embedded anchoring would be for pat- 2
terns such as `(^| )foo( |$)` to match `foo` when it exists as a complete word. 2
This functionality can be obtained using existing `lex` features: 2

```
^foo/[ \n]      |                                   2
" foo"/[ \n]    /* found foo as a separate word */  2
```

The precedence of regular expressions in `lex` does not match that of extended regular expressions in Section 2 because of historical practice. In System V `lex` and its predecessors, a regular expression of the form `ab{3}` matches `ababab`; an ERE, such as used by `egrep`, would match `abbb`. Changing this precedence for uniformity with `egrep` would have been desirable, but too many applications would break in nonobvious ways.

Conforming 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 recognized.

History of Decisions Made

Even though the `-c` option and references to the C language are retained in this description, `lex` may be generalized to other languages, as was done at one time for EFL, Extended FORTRAN Language. Since the `lex` input specification is essentially language independent, versions of this utility could be written to produce Ada, Modula-2, or Pascal code, and there are known historical implementations that do so.

The current description of `lex` bypasses the issue of dealing with internationalized regular expressions in the `lex` source code or generated lexical analyzer. If it follows the model used by `awk`, (the source code is assumed to be presented in the POSIX Locale, but input and output are in the locale specified by the environment variables), then the tables in the lexical analyzer produced by `lex` would interpret regular expressions specified in the `lex` source in terms of the environment variables specified when `lex` was executed. The desired effect would be to have the lexical analyzer interpret the regular expressions given in the `lex` source according to the environment specified when the lexical analyzer is executed, but this is not possible with the current `lex` technology.

Major international vendors believe that only limited internationalization is required for the POSIX.2 `lex`. The theoretically desirable goal of runtime-selectable locales is not feasible in the near future. Furthermore, the very nature of the lexical analyzers 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 analyzer that is used for French text will not automatically be useful for processing other languages.) The text in the Environment Variable subclause allows locale-specific regular expression handling, but mandates only something similar to that provided in historical implementations.

The description of octal- and hexadecimal-digit escape sequences agrees with the C Standard {7} use of escape sequences. See the rationale for `ed` for a discussion of bytes larger than nine bits being represented by octal values. Hexadecimal values can represent larger bytes and multibyte characters directly, using as many digits as required.

There is no detailed output format specification. The observed behavior of `lex` under four different historical implementations was that none of these implementations consistently reported the line numbers for error and warning messages. Furthermore, there was a desire that `lex` be allowed to output additional diagnostic messages. Leaving message formats unspecified sidesteps these formatting questions and also avoids problems with internationalization.

Although the `%x` specifier for `exclusive` start conditions is not existing practice, it is believed to be a minor change to historical implementations, and greatly enhances the usability of `lex` programs since it permits an application to obtain the expected functionality with fewer statements.

The `%array` and `%pointer` declarations were added as a compromise between historical systems. The System V-based `lex` has copied the matched text to a *yytext* array. The `flex` program, supported in BSD and GNU systems, uses a pointer. In the latter case, significant performance improvements are available for some scanners. Most existing programs should require no change in porting from one system to another because the string being referenced is null-terminated in both cases. (The method used by `flex` in its case is to null-terminate the token in-place by remembering the character that used to come right after the token and replacing it before continuing on to the next scan.) Multifile programs with external references to *yytext* outside the scanner source file should continue to operate on their existing systems, but would require one of the new declarations to be considered strictly portable.

The description of regular expressions avoids unnecessary duplication of regular expression details. Specifically, the **|** operator and {*m,n* } interval expression are not listed in A.2.7.4 because their meanings within a `lex` regular expression are the same as that for extended regular expressions.

The reason for the undefined condition associated with text beginning with a `<blank>` or within `%{` and `%}` delimiter lines appearing in the *Rules* section is historical practice. Both BSD and System V `lex` copy the indented (or enclosed) input in the *Rules* section (except at the beginning) to unreachable areas of the *yylex* () function (the code is written directly after a `break` statement). In some cases, the System V `lex` generates an error message or a syntax error, depending on the form of indented input.

The intention in breaking the list of functions into those that may appear in `lex.yy.c` versus those that only appear in `libl.a` is that only those functions in `libl.a` can be reliably redefined by a portable application.

The descriptions of Standard Output and Standard Error are somewhat complicated because historical `lex` implementations chose to issue diagnostic messages to standard output (unless `−t` was given). POSIX.2 allows this behavior, but leaves an opening for the more expected behavior of using standard error for diagnostics. Also, the System V behavior of writing the statistics when any table sizes are given is allowed, while BSD-derived systems can avoid it. The programmer can always precisely obtain the desired results by using either the `−t` or `−n` options.

The Operands subclause does not mention the use of – as a synonym for standard input; not all historical implementations support such usage for any of the *file* operands.

The description of the *Translation Table* was deleted from earlier drafts because of its relatively low usage in historical applications.

The change to the definition of the *input* () function that allows buffering of input presents the opportunity for major performance gains in some applications.

A.3 `yacc` — Yet another compiler compiler

A.3.1 Synopsis

`yacc [−dltv][−b` *file_prefix* `][−p` *sym_prefix* `]` *grammar*

A.3.2 Description

The `yacc` utility shall read a description of a context-free grammar in *file* and write C source code, conforming to the C Standard {7}, to a code file, and optionally header information into a header file, in the current directory. The C code shall define a function and related routines and macros for an automaton that executes a parsing algorithm meeting the requirements in A.3.7.8.

The form and meaning of the grammar is described in A.3.7.

The C source code and header file shall be produced in a form suitable as input for the C compiler (see `c89` in A.1).

A.3.3 Options

The `yacc` utility shall conform to the utility argument syntax guidelines described in 2.10.2.

The following options shall be supported by the implementation:

`−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), shall 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.

`−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.

`−p` *sym_prefix*
: Use *sym_prefix* instead of `yy` as the prefix for all external names produced by `yacc`. The names affected shall include the functions *yyparse* (), *yylex* (), and *yyerror* (), and the variables *yylval* , *yychar* , and *yydebug* . (In the remainder of this clause, 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 shall not affect `yacc`-generated `#define` symbols.

`−t` Modify conditional compilation directives to permit compilation of debugging code in the code file. Runtime debugging statements shall 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.

A.3.4 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 A.3.7.

A.3.5 External Influences

A.3.5.1 Standard Input

None.

A.3.5.2 Input Files

The file *grammar* shall be a text file formatted as specified in A.3.7.

A.3.5.3 Environment Variables

The following environment variables shall affect the execution of `yacc`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

The **LANG** and **LC_*** variables shall affect the execution of the `yacc` utility as stated. The *main* () function defined in A.3.7.6 shall call

```
setlocale(LC_ALL, "")
```

and thus, the program generated by `yacc` shall also be affected by the the contents of these variables at runtime.

A.3.5.4 Asynchronous Events

Default.

A.3.6 External Effects

A.3.6.1 Standard Output

None.

A.3.6.2 Standard Error

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.

A.3.6.3 Output Files

The code file, the header file, and the description file shall be text files. All are described in the following subclauses.

A.3.6.3.1 Code file

This file shall contain the C source code for the *yyparse* () routine. It shall contain code for the various semantic actions with macro substitution performed on them
as described in A.3.7. It shall also contain a copy of the `#define` statements in 2
the header file. If a `%union` declaration is used, the declaration for YYSTYPE 2
shall be also included in this file. 2

The contents of the Program Section (see A.3.7.1.4) of the input file shall then be included.

A.3.6.3.2 Header file

The header file shall 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 shall be also included in this file.

A.3.6.3.3 Description file

The description file shall be a text file containing a description of the state machine corresponding to the parser, using an unspecified format. Limits for internal tables (see A.3.7.9) also shall be reported, in an implementation-defined manner.

A.3.7 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 recognized to provide the associated semantic action. The code to be executed shall 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 shall be executed during the recognition of the target language.

Given a grammar, the `yacc` utility generates the files described in A.3.6.3. The code file can be compiled and linked using `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 `c89`. The `yacc` library interfaces need not support interfaces with other than the default `yy` symbol prefix. The application provides the lexical analyzer function, *yylex* (); the `lex` utility (see A.2) is specifically designed to generate such a routine.

A.3.7.1 Input Language

Every specification file shall consist of three sections: *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.

A.3.7.1.1 Lexical Structure of the Grammar

The characters `<blank>`s, `<newline>`s, and `<form-feed>`s shall be ignored, except that they shall not appear in names or multicharacter reserved symbols. Comments shall be enclosed in `/* ... */`, and can appear wherever a name is valid.

Names are of arbitrary length, made up of letters, periods (`.`), underscores (`_`), and noninitial digits. Upper- and lowercase letters are distinct. Portable applications shall 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 shall consist of a single character enclosed in single-quotes (′). All of the escape sequences supported for character constants by the C Standard {7}(3.1.3.4) shall be supported by `yacc`.

The relationship with the lexical analyzer is discussed in detail below.

The NUL character shall not be used in grammar rules or literals.

A.3.7.1.2 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 standard, 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 shall 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 shall be declared to be the type referenced by *tag*. If a positive integer, *number*, follows a *name*, that value shall 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 shall declare 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 shall report an error. If *tag* is present, it shall declare a C type for *name* (s) as described for `%token`.

`%type` <*tag*> *name* ...
: Declares that union member *name* (s) are nonterminals, and thus it is required to have a *tag* field at its beginning. Because it deals with nonterminals only, assigning a token number or using a literal is also prohibited. If this construct is present, `yacc` shall perform type checking; if this construct is not present, the parse stack shall hold only the `int` type.

Every name used in *grammar* undefined by a `%token`, `%left`, `%right`, or `%nonassoc` declaration is assumed to represent a nonterminal symbol. The `yacc` utility shall report an error for any nonterminal 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 shall not be changed. If the first declaration of a token does not assign a token number, `yacc` shall assign a token number. Once this assignment is made, the token number shall not be changed by explicit assignment.

The following declarators do not follow the previous pattern.

`%start` *name*
: Declares the nonterminal *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 analyzer shall be integers. The `yacc` utility keeps track of types, and shall 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 shall 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 shall 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 shall be terminated by the token `%%`.

A.3.7.1.3 Grammar Rules

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 nonterminal 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 nonterminal 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 *nonterminal* ; *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 recognized in the input process. [Note that the word ‘‘action’’ can also refer to the actions of the parser (shift, reduce, etc.).]

These actions can return values and can obtain the values returned by previous actions. These values shall be kept in objects of type YYSTYPE (see `%union`). The result value of the action shall 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 analyzer 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 interally 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 1
not positive. 1

`$`<*tag*>`$`

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 shall be equivalent to replacing the action with a new nonterminal symbol and adding an empty rule with that nonterminal symbol on the left-hand side. The semantic action associated with the new rule shall 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 shall be 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 shall result.

precedence The keyword `%prec` can be used to change the precedence level 1
associated with a particular grammar rule. Examples of this are 1
in cases where a unary and binary operator have the same sym- 1
bolic representation, but need to be given different precedences, 1
or where the handling of an ambiguous if-else construction is 1
necessary. The reserved symbol `%prec` can appear immediately 1
after the body of the grammar rule and can be followed by a token name or a literal. It shall 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 shall be terminated by %%. 1

A.3.7.1.4 Programs Section

The *programs* section can include the definition of the lexical analyzer *yylex* (), and any other functions, for example those used in the actions specified in the grammar rules. This is C language code, and shall 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 shall place them within `%{ ... %}` in the declarations section.

A.3.7.1.5 Input Grammar

The following input to `yacc` yields a parser for the input to `yacc`. This is to be taken as the formal specification of the grammar of `yacc`, notwithstanding conflicts that may appear elsewhere.

The lexical structure is defined less precisely; the previous section on A.3.7.1.1 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 nonnegative decimal integer).

`TYPE`
`LEFT`
`MARK`
etc. These correspond directly to `%type`, `%left`, `%%`, etc.

`{ ... }` 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 recognized by the lexical analyzer */

%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, etc. */

%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 multibyte 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 $$, etc. */
          }
          '}'
        ;

prec  : /* empty */
        | PREC IDENTIFIER
        | PREC IDENTIFIER act
        | prec ';'
        ;
```

A.3.7.2 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 shall 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 than was intended. The description file shall 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 A.3.7.1.2) 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 favor of the action (shift or reduce) associated with the higher precedence. If the precedences are the same, then the associativity is used; left associative implies reduce, right associative implies shift, and nonassociative implies 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 A.3.7.3.

(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 A.3.7.3.

Conflicts resolved by precedence or associativity shall not be counted in the shift/reduce and reduce/reduce conflicts reported by `yacc` on either standard error or in the description file.

A.3.7.3 Error Handling

The token `error` shall be 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` shall be 256. Its value can be changed using a `%token` declaration. The lexical analyzer should not return the value of `error`.

The parser shall 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 shall 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 shall 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 shall not call `yyerror` when YYERROR is executed.

The macro function YYRECOVERING () shall return 1 if a syntax error has been detected and the parser has not yet fully recovered from it. Otherwise, zero shall be returned.

When a syntax error is detected by the parser, the parser shall 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 A.3.7.4). If it is, the parser shall return with a nonzero value. Otherwise, the lookahead symbol shall be discarded and normal parsing shall 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 shall 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 shall return with a nonzero value. Otherwise, it shall shift over `error` and then resume normal parsing. If the parser reads a lookahead symbol before the error was detected, that symbol shall still be the lookahead symbol when parsing is resumed.

The macro `yyerrok` in a semantic action shall cause the parser to act as if it has fully recovered from any previous errors. The macro `yyclearin` shall cause the parser to discard the current lookahead token. If the current lookahead token has not yet been read, `yyclearin` shall have no effect.

The macro YYACCEPT shall cause the parser to return with the value zero. The macro YYABORT shall cause the parser to return with a nonzero value.

A.3.7.4 Interface to the Lexical Analyzer

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 shall 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 shall not be included in the `#define` declarations.

If the token numbers are chosen by `yacc`, the tokens other than literals shall be assigned numbers greater than 256, although no order is implied. A token can be 1
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 shall be unique and distinct from the token numbers used for literals. If duplicate token numbers cause conflicts in parser generation, `yacc` shall 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 analyzers shall 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 shall accept the input. If the endmarker is seen in any other context, it shall be considered an error.

A.3.7.5 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.

A.3.7.6 `yacc` Library

The following functions appear only in the `yacc` library accessible through the `-l y` operand to `c89`; they can therefore be redefined by a portable application:

`int main(void)` 1

This function shall call *yyparse* () and exit with an unspecified value. Other actions within this function are unspecified.

`int yyerror(const char *s)` 1

This function shall write the NUL-terminated argument to standard error, followed by a <newline>.

The order of the −l y and −l l operands given to c89 is significant; the application shall either provide its own *main* () function or ensure that −l y precedes −l l.

A.3.7.7 Debugging the Parser

The parser generated by yacc shall 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 nonzero value, the debugging code shall be included. If its value is zero, the code shall 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 nonzero value) and off (zero value) at run time. The initial value of *yydebug* shall be zero.

When −t is specified, the code file shall be built such that, if YYDEBUG is not already defined at compilation time (using the c89 -D YYDEBUG option, for example), YYDEBUG shall be set explicitly to 1. When −t is not specified, the code file shall be built such that, if YYDEBUG is not already defined, it shall 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.

A.3.7.8 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 shall 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.

A.3.7.9 Limits

The yacc utility may have several internal tables. The minimum maximums for these tables are shown in Table A-4. The exact meaning of these values is implementation defined. The implementation shall 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 subclause.

Table A-4 – `yacc` Internal Limits

Limit	Minimum Maximum	Description
{NTERMS }	126	Number of tokens.
{NNONTERM }	200	Number of nonterminals.
{NPROD }	300	Number of rules.
{NSTATES }	600	Number of states.
{MEMSIZE }	5200	Length of rules. The total length, in names (tokens and nonterminals), 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 A.3.7.1.3.
{ACTSIZE }	4000	Number of actions. ''Actions'' here (and in the description file) refer to parser actions (shift, reduce, etc.) not to semantic actions defined in A.3.7.1.3.

A.3.8 Exit Status

The `yacc` utility shall exit with one of the following values:

- 0 Successful completion.
- >0 An error occurred.

A.3.9 Consequences of Errors

If any errors are encountered, the run is aborted and `yacc` exits with a nonzero status. Partial code files and header files files may be produced. The summary information in the description file shall always be produced if the `-v` flag is present.

A.3.10 Rationale. *(This subclause is not a part of P1003.2)*

The references in the Bibliography may be helpful in constructing the parser generator. The Pennello-DeRemer {B26} paper (along with the works it references) 2
describe a technique to generate parsers that conform to this standard. Work in this area continues to be done, so implementors should consult current literature before doing any new implementations. The original paper by Knuth {B27} is the 1
theoretical basis for this kind of parser, but the tables it generates are impractically large for reasonable grammars, and should not be used. The ''equivalent to'' wording is intentional to assure that the best tables that are LALR(1) can be generated.

There has been confusion between the class of grammars, the algorithms needed to generate parsers, and the algorithms needed to parse the languages. They are all reasonably orthogonal. In particular, a parser generator that accepts the full range of LR(1) grammars need not generate a table any more complex than one that accepts SLR(1) (a relatively weak class of LR grammars) for a grammar that happens to be SLR (1). Such an implementation need not recognize the case, either; table compression can yield the SLR (1) table (or one even smaller than that) without recognizing that the grammar is SLR (1). The speed of a LR (1) parser for any class is dependent more upon the table representation and compression (or the code generation if a direct parser is generated) than upon the class of grammar that the table generator handles.

The speed of the parser generator is somewhat dependent upon the class of grammar it handles. However, the original Knuth {B27} algorithms for constructing 2
LR parsers was judged by its author to be impractically slow at that time. 2
Although full LR is more complex than LALR (1), as computer speeds and algorithms improve, the difference (in terms of acceptable wall-clock execution time) is 2
becoming less significant. 2

Potential authors are cautioned that the Penello-DeRemer paper previously cited 2
identifies a bug (an oversimplification of the computation of LALR (1) lookahead 2
sets) in some of the LALR (1) algorithm statements that preceded it to publication. 2
They should take the time to seek out that paper, as well as current relevant 2
work, particularly Aho's {B22}.

Examples, Usage

Access to the `yacc` library is obtained with library search operands to `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);
}
```

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 hardcoded `yy` variable prefix. (The `−p` option name was selected from a historical implementation.) Implementations will also have to be cognizant of 2.11.6.3, which requires that any temporary files used by `yacc` also be named to avoid collisions.

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* don't 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`.

The `−b` option was added to provide a portable method for permitting `yacc` to work on multiple separate parsers in the same directory. If a directory contains more than one `yacc` grammar, and both grammars are constructed at the same time (by, say, a parallel `make` program), conflict results. While the solution is not historical practice, it corrects a known deficiency in historical implementations. Corresponding changes were made to all sections that referenced the filenames `y.tab.c` (now ''the code file''), `y.tab.h` (now ''the header file''), and `y.output` (now ''the description file'').

The grammar for `yacc` input is based on System V documentation. The textual description shows there that the `;` is required at the end of the rule. The grammar and the implementation do not require this. (The use of `C_IDENTIFIER` causes a reduce to occur in the right place.)

Also, in that implementation, the constructs such as `%token` can be terminated by a semicolon, but this is not permitted by the grammar. The keywords such as `%token` can also appear in uppercase, which is again not discussed. In most places where `%` is used, `\` can be substituted, and there are alternate spellings for some of the symbols (e.g. `%LEFT` can be `%<` or even `\<`).

Multibyte characters should be recognized by the lexical analyzer and returned as tokens. They should not be returned as multibyte 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 multibyte character sets, is not available for use as the value of a user-defined token.

Historically, *<tag>* can contain any characters except `>`, including white space, in the implementation. However, since the *tag* must reference a Standard C union member, in practice conforming implementations need only support the set of characters for Standard C identifiers in this context.

Some historical implementations are known to accept actions that are terminated by a period. Historical implementations often allow `$` in names. A conforming implementation need support neither of these behaviors.

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 A. Applications are not required to use this operator for unary operators, but the grammars that do not require it are rare.

Deciding when to use `%prec` illustrates the difficulty in specifying the behavior of `yacc`. There may be situations in which the *grammar* is not strictly speaking in error, and yet `yacc` cannot interpret it unambiguously. The resolution of ambiguities in the grammar can in many instances be resolved by providing additional information, such as using `%type` or `%union` declarations. It is often easier and it usually yields a smaller parser to take this alternative when it is appropriate.

The size and execution time of a program produced without the runtime debugging code is usually smaller and slightly faster in historical implementations.

There is a fair amount of material in this that appears tutorial in nature; some of it has been moved to the Rationale in Draft 9 to simplify the specification. It is hard to avoid because of the need to define terms at least informally. The alternative is to bring in one of the parser generator texts and use its terminology directly, but since there is some variation in that terminology, it was felt that informal definitions of the terms so that someone who understood the concepts would be sure to understand the terms would make the standard stand alone from any specific text.

Statistics messages from several historical implementations include the following types of information:

n /512 terminals, *n* /300 nonterminals
n /600 grammar rules, *n* /1500 states
n shift/reduce, *n* reduce/reduce conflicts reported
n /350 working sets used
memory: states,etc. *n* /15000, parser *n* /15000
n /600 distinct lookahead sets
n extra closures
n shift entries, *n* exceptions
n goto entries
n entries saved by goto default
Optimizer space used: input *n* /15000, output *n* /15000
n table entries, *n* zero
maximum spread: *n*, maximum offset: *n*

The report of internal tables in the description file is left implementation defined because all aspects of these limits are also implementation defined. Some implementations may use dynamic allocation techniques and have no specific limit values to report. 2

History of Decisions Made

The format of the `y.output` file is not given because specification of the format was not seen to enhance application portability. The listing is primarily intended to help human users understand and debug the parser; use of `y.output` by a portable application script is far-fetched. Furthermore, implementations have not produced consistent output and no clear winner was apparent. The format selected by the implementation should be human-readable, in addition to the requirement that it be a text file.

Standard error reports are not specifically described because they are seldom of use to portable applications and there was no reason to restrict implementations.

Some implementations recognize `={` as equivalent to `{`, because it appears in historical documentation. This construction was recognized and documented as obsolete as long ago as 1978, in the original paper *Yacc: Yet Another Compiler-Compiler* by Stephen C. Johnson. POSIX.2 chose to leave it as obsolete and omit it.

Annex B
(norm ative)

C Language Bindings Option

This annex describes the C language bindings to the language-independent services described in Section 7.

The interfaces described in this annex may be provided by the conforming system; however, any system claiming conformance to the Language-Independent System Services C Language Bindings Option shall provide all of the interfaces described here.

B.0.1 C Language Bindings Option Rationale. *(This subclause is not a part of P1003.2)*

In this version of POSIX.2, the language-independent descriptions in Section 7 have not been developed. The language-independent syntax is being created in parallel by the POSIX.1 working group. Therefore, the C language bindings described in this annex are actually the full functional specifications. It is the intention of the POSIX.2 working group to rectify this situation in a revision to this standard, by moving the majority of the functional specifications back into Section 7, leaving Annex B with only brief descriptions of the C bindings to those services.

B.1 C Language Definitions

B.1.1 POSIX Symbols

Certain symbols in this annex are defined in headers. Some of those headers could also define symbols other than those defined by this standard, potentially conflicting with symbols used by the application. Also, this standard defines symbols that other standards do not permit to appear in those headers without some control on the visibility of those symbols.

Symbols called *feature test macros* are used to control the visibility of symbols that might be included in a header. Implementations, future versions of this standard, and other standards may define additional feature test macros. The `#define`s for feature test macros shall appear in the application source code before any `#include` of a header where a symbol should be visible to some, but not all, applications. If the definition of the macro does not precede the `#include`, the result is undefined.

Feature test macros shall begin with the underscore character (_) and an uppercase letter, or with two underscore characters.

Implementations may add symbols to the headers shown in Table B-1, provided the identifiers for those symbols begin with the corresponding reserved prefixes in Table B-1. Similarly, implementations may add symbols to the headers in Table B-1 that end in the string indicated as a reserved suffix as long as the reserved suffix is in that part of the name considered significant by the implementation. This shall be in addition to any reservations made in the C Standard {7}.

After the last inclusion of a given header, an application may use any of the symbol classes reserved in Table B-1 for its own purposes, as long as the requirements in the note to Table B-1 are satisfied, noting that the symbol declared in the header may become inaccessible.

Future revisions of this standard, and other POSIX standards, are likely to use symbols in these same reserved spaces.

In addition, implementations may add members to a structure or union without controlling the visibility of those members with a feature test macro, as long as a user-defined macro with the same name cannot interfere with the correct interpretation of the program.

A conforming POSIX.2 application shall define the feature test macro in Table B-2. When an application includes a header and the _POSIX_C_SOURCE feature test macro is defined to be the value 1 or 2, the effect shall be the same as if _POSIX_SOURCE was defined as described in POSIX.1 {8}.

In addition, when the application includes any of the headers defined in this standard, and _POSIX_C_SOURCE is defined to be the value 2:

(1) All symbols defined in POSIX.2 to appear when the header is included shall be made visible.

Table B-1 – POSIX.2 Reserved Header Symbols

Header	Key	Reserved Prefix	Reserved Suffix
`<fnmatch.h>`	2	`FNM_`	
`<glob.h>`	1	`gl_`	
	2	`GLOB_`	
`<limits.h>`	1		`_MAX`
`<regex.h>`	1	`re_`	
	1	`rm_`	
	2	`REG_`	
`<wordexp.h>`	1	`we_`	
	2	`WRDE_`	

NOTE: The Key values are:

(1) Prefixes and suffixes of symbols that shall not be declared or `#define`d by the application.

(2) Prefixes and suffixes of symbols that shall be preceded in the application with a `#undef` of that symbol before any other use.

Table B-2 – _POSIX_C_SOURCE

Name	Description
_POSIX_C_SOURCE	Enable POSIX.1 {8} and POSIX.2 symbols; see text.

(2) Symbols that are explicitly permitted, but not required, by POSIX.2 to appear in the header (including those in reserved name spaces) may be made visible.

(3) Additional symbols shall not be made visible, unless controlled by another feature test macro.

The effect of defining the _POSIX_C_SOURCE macro to any other value is unspecified.

If there are no feature test macros present in a program, only the set of symbols defined by the C Standard {7} shall be present. For each feature test macro present, only the symbols specified by that feature test macro plus those of the C Standard {7} shall be defined when the header is included.

B.1.1.1 POSIX Symbols Rationale. *(This subclause is not a part of P1003.2)*

When the application defines the _POSIX_C_SOURCE feature test macro with value 2, it must be aware that all of the name space from POSIX.1 {8} and POSIX.2 has been reserved. This does not imply that a POSIX.2 implementation must support POSIX.1 {8}, just that the application must not conflict with an implementation that does. The application can check _POSIX_VERSION and _POSIX2_C_VERSION at compile time to see which standards are supported, if that is necessary. This is primarily an issue for the headers `<stdio.h>`, `<limits.h>`, `<locale.h>`, and `<unistd.h>`, since other POSIX.1 {8} names appear in other headers not mentioned in POSIX.2.

It is expected that C bindings to future POSIX standards and revisions will define new values for _POSIX_C_SOURCE, with each new value reserving the name space for that new standard or revision, plus all earlier POSIX standards. Using a single feature test macro for all standards rather than a separate macro for each standard furthers the goal of eventually combining all of the C bindings into one standard, which will be included in an international standard that refers to a language-independent ISO/IEC 9945-1 {8}.

B.1.2 Headers and Function Prototypes

Implementations shall declare function prototypes for all functions. Each function prototype shall appear in the header included in the synopsis of the function.

B.1.3 Error Numbers

Some of the functions in this annex use the variable *errno* to report errors. Such usage is documented in Errors in each specification. The usage of *errno* and the meanings of the symbolic names shall be as defined in POSIX.1 {8} B.1.3.

B.1.4 C Language Definitions Rationale. *(This subclause is not a part of P1003.2)*

This clause clarifies the interface to the C Standard {7}. The description was taken from POSIX.1, with one important modification. Since POSIX.1 {8} and the C Standard {7} were being developed and approved at about the same time, POSIX.1 {8} allowed ‘‘Common Usage C’’ implementations to give system vendors time to develop Standard C interfaces. Since Standard C compilers are now commonly available, POSIX.2 does not explicitly describe the binding to Common Usage C. However, such a binding would be straightforward, as long as the rules for Common Usage C in POSIX.1 are followed.

B.2 C Numerical Limits

The following subclauses list the names of macros that C language applications can use to obtain minimum and current values for limits defined in 2.13.1.

B.2.0.1 C Numerical Limits Rationale. *(This subclause is not a part of P1003.2)*

This subclause was added in Draft 9 to give C applications access to limits at compile time. Applications can use the values from the macros without resorting to *sysconf* (). The descriptions very closely follow the descriptions of macros and limits in POSIX.1 {8}.

This definition of the limits is specific to the C language. Other language bindings might use different interfaces or names to provide equivalent information to the application.

Note that there are no C bindings or interfaces that change based on the macros in Table B-5. These macro only advertise the availability of the associated utilities.

B.2.1 C Macros for Symbolic Limits

The macros in Table B-3 shall be defined in the header `<limits.h>`. They specify values for the symbolic limits defined in 2.13.1.

Table B-3 – C Macros for Symbolic Limits

Symbolic Limit	Minimum Allowed by POSIX.2	Minimum for this Implementation
{BC_BASE_MAX}	_POSIX2_BC_BASE_MAX	BC_BASE_MAX
{BC_DIM_MAX}	_POSIX2_BC_DIM_MAX	BC_DIM_MAX
{BC_SCALE_MAX}	_POSIX2_BC_SCALE_MAX	BC_SCALE_MAX
{BC_STRING_MAX}	_POSIX2_BC_STRING_MAX	BC_STRING_MAX
{COLL_WEIGHTS_MAX}	_POSIX2_COLL_WEIGHTS_MAX	COLL_WEIGHTS_MAX
{EXPR_NEST_MAX}	_POSIX2_EXPR_NEST_MAX	EXPR_NEST_MAX
{LINE_MAX}	_POSIX2_LINE_MAX	LINE_MAX
{RE_DUP_MAX}	_POSIX2_RE_DUP_MAX	RE_DUP_MAX

The names in the first column of Table B-3 are symbolic limits as defined in 2.13.1. The names in the second column are C macros that define the smallest values permitted for the symbolic limits on any POSIX.2 implementation; they shall be defined as constant expressions with the most restrictive values specified in 2.13.1. The names in the third column are C macros that define less restrictive values provided by the implementation; each shall be defined as a constant that

— is not smaller than the associated macro in column 2, and

— is not larger than the smallest value that will be returned by *sysconf* () when the application is executed.

B.2.1.1 C Macros for Symbolic Limits Rationale. *(This subclause is not a part of P1003.2)*

The macros in column 3 of Table B-3 are required to be constant expressions.

If the C binding is to be used with POSIX.2 implementations over which the implementor of the binding has no control, the column-3 values must be the same as column-2. If the implementation of the C binding is intended to be used with a POSIX.2 implementation that always supports a larger value than one in column 2, that implementation of the binding may use the larger value for the column-3 macro. If an application compiled with that binding is then used with a different POSIX.2 implementation, it is the user's fault that the application is being run in an environment in which it was not intended.

The application can assume, for example, that the stream created by `popen("mailx user","w")` will accept lines of length {LINE_MAX}, even if this is larger than {_POSIX2_LINE_MAX}. However, if the application is creating a data file that might be processed on another implementation, it should use the values in column 2.

B.2.2 Compile-Time Symbolic Constants for Portability Specifications

The macros in Table B-4 shall be defined in the header `<unistd.h>`. These macros can be used by the application, at compile time, to determine which optional facilities are present and what actions shall be taken by the implementation.

Table B-4 – C Compile-Time Symbolic Constants

Macro Name	Description
_POSIX2_C_VERSION	The integer value 199???L. This value indicates the version of the interfaces in this annex that are provided by the implementation. It will change with each published version of this standard to indicate the 4-digit year and 2-digit month that the standard was approved by the IEEE Standards Board.

B.2.2.1 Compile-Time Symbolic Constants for Portability Specifications Rationale. *(This subclause is not a part of P1003.2)*

The symbolic constant _POSIX2_C_VERSION is analogous to _POSIX_VERSION, defined in POSIX.1 {8}. It indicates the version of the C interfaces that are supplied by the compiler and runtime library.

The version of the utilities is given by the {POSIX2_VERSION} limit (see 2.13.1), whose value can be obtained at runtime using *sysconf*() (see B.10.2).

B.2.3 Execution-Time Symbolic Constants for Portability Specifications

The macros in Table B-5 can be used by the application at execution time to determine which optional facilities are present. If a macro is defined to have the value −1 in the header `<unistd.h>`, the implementation shall never provide that feature when the application runs under that implementation. If a macro is defined to have a value other than −1, the implementation shall always provide that feature. If the macro is undefined, then the *sysconf* () function (see B.10.2) can be used to determine if the feature is provided for a particular invocation of the application.

Table B-5 − C Execution-Time Symbolic Constants

Macro Name	Description
_POSIX2_C_DEV	The system supports the C Language Development Utilities Option (see Annex A)
_POSIX2_FORT_DEV	The system supports the FORTRAN Development Utilities Option (see Annex C)
_POSIX2_FORT_RUN	The system supports the FORTRAN Runtime Utilities Option (see Annex C)
_POSIX2_LOCALEDEF	The system supports the creation of locales as described in 4.35.
_POSIX2_SW_DEV	The system supports the Software Development Utilities Option (see Section 6)

B.2.4 POSIX.1 C Numerical Limits

The macros specified in POSIX.1 {8} to provide compile-time values for the configurable variables in Table 7-1 (see 7.8.2) shall also be visible in a POSIX.2 system. Other macros required by POSIX.1 {8} 2.9 (Numerical Limits) and 2.10 (Symbolic Constants) may also be visible in a POSIX.2 system.

B.2.4.1 POSIX.1 C Numerical Limits Rationale. *(This subclause is not a part of P1003.2)*

Subclause 7.8.2 requires that certain POSIX.1 {8} configurable variables be visible in POSIX.2. Subclause B.2.4 ensures that POSIX.2 C applications can obtain these variables using the same macros as POSIX.1 {8} C applications. It also allows an implementation to make all of the POSIX.1 {8} macros available even if
_POSIX_SOURCE is not set. It also allows an implementation to make all of the 1
POSIX.1 {8} symbols available even if it does not support all of POSIX.1 {8}. 1

B.3 C Binding for Shell Command Interface

B.3.0.1 C Binding for Shell Command Interface Rationale. *(This subclause is not a part of P1003.2)*

The *system* () and *popen* () functions should not be used by programs that have set user (or group) ID privileges, as defined in POSIX.1 {8}. The *fork* () and *exec* family of functions [except *execlp* () and *execvp* ()], also defined in POSIX.1 {8}, should be used instead. This prevents any unforeseen manipulation of the user's environment that could cause execution of commands not anticipated by the calling program.

If the original and '*popen* ()ed'' processes both intend to read or write or read and write a common file, and either will be using FILE-type C functions [*fread* (), *fwrite* (), etc.], the rules in POSIX.1 {8}8.2.3 must be observed.

B.3.1 C Binding for Execute Command

Function: *system* ()

B.3.1.1 Synopsis

```
#include <stdlib.h>
int system(const char *command );
```

B.3.1.2 Description

This standard requires the *system* () function as described in the C Standard {7}.

The *system* () function shall execute the command specified by the string pointed to by *command* . The environment of the executed command shall be as if a child process were created using the POSIX.1 {8} *fork* () function, and the child process invoked the sh utility (see 4.56) using the POSIX.1 {8} *execl* () function as follows:

```
execl(<shell path>, "sh", "-c", command , (char *)0);
```

where *<shell path>* is an unspecified pathname for the sh utility.

The *system* () function shall ignore the SIGINT and SIGQUIT signals, and block the SIGCHLD signal, while waiting for the command to terminate. If this might cause the application to miss a signal that would have killed it, then the application should examine the return value from *system* () and take whatever action is appropriate to the application if the command terminated due to receipt of a signal.

The *system* () function shall not affect the termination status of any child of the calling processes other than the process(es) it itself creates.

The *system* () function shall not return until the child process has terminated.

B.3.1.3 Returns

If *command* is **NULL**, the *system*() function shall return nonzero.

If *command* is not **NULL**, the *system*() function shall return the termination status of the command language interpreter in the format specified by the *waitpid*() function in POSIX.1 {8}. The termination status of the command language interpreter is as specified for the sh utility, except that if some error prevents the command language interpreter from executing after the child process is created, the return value from *system*() shall be as if the command language interpreter had terminated using *exit*(127) or *_exit*(127). If a child process cannot be created, or if the termination status for the command language interpreter cannot be obtained, *system*() shall return −1 and set *errno* to indicate the error.

B.3.1.4 Errors

The *system*() function may set *errno* values as described by *fork*() in POSIX.1 {8}.

B.3.1.5 Rationale. *(This subclause is not a part of P1003.2)*

The C Standard {7} specifies that when *command* is **NULL**, *system*() returns nonzero if there is a command interpreter available and zero if one is not available. At first reading, it might appear that POSIX.2 conflicts with this, since it requires *system*(**NULL**) to always return nonzero. There is no conflict, however. A POSIX.2 implementation must always have a command interpreter available, and is nonconforming if none is present. It is therefore permissible for the *system*() function on a POSIX.2 system to implement the behavior specified by the C Standard {7} as long as it is understood that the implementation is not POSIX.2 conforming if *system*(**NULL**) returns zero. 1

Note that, while *system*() must ignore SIGINT and SIGQUIT and block SIGCHLD while waiting for the child to terminate, the handling of signals in the executed command is as specified by *fork*() and *exec*. For example, if SIGINT is being caught or is set to SIG_DFL when *system*() is called, then the child will be started with SIGINT handling set to SIG_DFL.

Ignoring SIGINT and SIGQUIT in the parent process prevents coordination problems (two processes reading from the same terminal, for example) when the executed command ignores or catches one of the signals. It is also usually the correct action when the user has given a command to the application to be executed synchronously (as in the ‘‘!’’ command in many interactive applications). In either case, the signal should be delivered only to the child process, not to the application itself. There is one situation where ignoring the signals might have less than the desired effect. This is when the application uses *system*() to perform some task invisible to the user. If the user typed the interrupt character (^C for example) while *system*() is being used in this way, one would expect the application to be killed, but only the executed command will be killed. Applications that use *system*() in this way should carefully check the return status from *system*() to see if the executed command was successful, and should take appropriate action when the command fails.

Blocking SIGCHLD while waiting for the child to terminate prevents the application from catching the signal and obtaining status from *system* ()'s child process before *system* () can get the status itself.

Examples, Usage

The context in which the utility is ultimately executed may differ from that in which the *system* () function was called. For example, file descriptors that have the FD_CLOEXEC flag set will be closed, and the process ID and parent process ID will be different. Also, if the executed utility changes its environment variables or its current working directory, that change will not be reflected in the caller's context.

Earlier drafts of this standard required, or allowed, *system* () to return with *errno* [EINTR] if it was interrupted with a signal. This error return was removed, and a requirement that *system* () not return until the child has terminated was added. This means that if a *waitpid* () call in *system* () exits with *errno* [EINTR], *system* () must re-issue the *waitpid* (). This change was made for two reasons:

(1) There is no way for an application to clean up if *system* () returns [EINTR], short of calling *wait* (), and that could have the undesirable effect of returning status of children other than the one started by *system* ().

(2) While it might require a change in some historical implementations, those implementations already have to be changed because they use *wait* () instead of *waitpid* ().

Note that if the application is catching SIGCHLD signals, it will receive such a sig- 1
nal before a successful *system* () call returns. 1

History of Decisions Made

The C Standard {7} requires that a call to *system* () with a **NULL** will return a nonzero value, indicating the presence of a command language interpreter available to the system. It was explicitly decided that when *command* is **NULL** , *system* () should not be required to check to make sure that the command language interpreter actually exists with the correct mode, that there are enough processes to execute it, etc. The call *system* (**NULL**) could, theoretically, check for such problems as too many existing child processes, and return zero. However, it would be inappropriate to return zero due to such a (presumably) transient condition. If some condition exists that is not under the control of this application and that would cause *any* *system* () call to fail, that system has been rendered nonconformant.

Modified in Draft 6 to reflect the availability of the *waitpid* () function in POSIX .1 {8}. To conform to this standard, *system* () must use *waitpid* (), or some similar function, instead of *wait* ().

Figure B-1 illustrates how *system* () might be implemented on a POSIX .1 {8} implementation.

```
#include <signal.h>

int system(const char *cmd)
{
    int     stat;
    pid_t   pid;
    struct sigaction sa, savintr, savequit;
    sigset_t saveblock;

    if (cmd == NULL)
        return(1);
    sa.sa_handler = SIG_IGN;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0;
    sigemptyset(&savintr.sa_mask);
    sigemptyset(&savequit.sa_mask);
    sigaction(SIGINT, &sa, &savintr);
    sigaction(SIGQUIT, &sa, &savequit);
    sigaddset(&sa.sa_mask, SIGCHLD);
    sigprocmask(SIG_BLOCK, &sa.sa_mask, &saveblock);
    if ((pid = fork()) == 0) {
        sigaction(SIGINT, &savintr, (struct sigaction *)0);
        sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
        sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
        execl("/bin/sh", "sh", "-c", cmd, (char *)0);
        _exit(127);
    }
    if (pid == -1) {
        stat = -1;  /* errno comes from fork() */
    } else {
        while (waitpid(pid, &stat, 0) == -1) {
            if (errno != EINTR) {
                stat = -1;
                break;
            }
        }
    }
    sigaction(SIGINT, &savintr, (struct sigaction *)0);
    sigaction(SIGQUIT, &savequit, (struct sigaction *)0);
    sigprocmask(SIG_SETMASK, &saveblock, (sigset_t *)0);
    return(stat);
}
```

Figure B-1 – Sample *system* () Implementation

Note that, while a particular implementation of *system* () (such as the one above) can assume a particular path for the shell, such a path is not necessarily valid on another system. The above example is not portable, and is not intended to be. There is no defined way for an application to find the specific path for the shell. However, *confstr* () can provide a value for **PATH** that is guaranteed to find the sh utility.

One reviewer suggested that an implementation of *system* () might want to use an environment variable such as **SHELL** to determine which command interpreter to use. The supposed implementation would use the default command interpreter if the one specified by the environment variable was not available. This would allow a user, when using an application that prompts for command lines to be processed using *system* (), to specify a different command interpreter. Such an implementation is discouraged. If the alternate command interpreter did not follow the command line syntax specified in POSIX.2, then changing **SHELL** would render *system* () nonconformant. This would affect applications that expected the specified behavior from *system* (), and since this standard does not mention that **SHELL** affects *system* (), the application would not know that it needed to unset **SHELL** .

B.3.2 C Binding for Pipe Communications with Programs

Functions: *popen* (), *pclose* ()

B.3.2.1 Synopsis

```
#include <stdio.h>

FILE *popen(const char *command , const char *mode );

int pclose(FILE *stream );
```

B.3.2.2 Description

The *popen* () function shall execute the command specified by the string *command* . It shall create a pipe between the calling program and the executed command, and return a pointer to a C Standard {7} stream that can be used to either read from or write to the pipe. The *pclose* () function shall close the stream, wait for the command to terminate, and return the termination status from the command language interpreter.

The environment of the executed command shall be as if a child process were created within the *popen* () call using the *fork* () function, and the child invoked the `sh` utility using the call:

```
execl(<shell path >, "sh", "-c", command , (char *)0);
```

where *<shell path>* is an unspecified pathname for the `sh` utility. However, 1
popen () shall ensure that any streams from previous *popen* () calls that remain 1
open in the parent process are closed in the new child process. 1

The *mode* argument to *popen* () is a string that specifies I/O mode:

(1) If *mode* is `"r"`, when the child process is started its file descriptor STDOUT_FILENO shall be the writable end of the pipe, and the file descriptor *fileno* (*stream*) in the calling process, where *stream* is the stream pointer returned by *popen* (), shall be the readable end of the pipe.

(2) If *mode* is `"w"`, when the child process is started its file descriptor STDIN_FILENO shall be the readable end of the pipe, and the file descriptor *fileno*(*stream*) in the calling process, where *stream* is the stream pointer returned by *popen*(), shall be the writable end of the pipe.

(3) If *mode* is any other value, the result is undefined.

A stream opened by *popen*() should be closed by *pclose*(). As stated above, *pclose*() shall return the termination status from the command language interpreter. However, if the application has called any of the following:

(1) *wait*(),

(2) *waitpid*() with a *pid* argument less than or equal to zero or equal to the process ID of the command line interpreter, or

(3) any other function not defined in POSIX.1 {8} or POSIX.2 that could do one of the above

and one of those calls caused the termination status to be unavailable to *pclose*(), then *pclose*() shall return −1 with *errno* set to [ECHILD] to report this situation. In any case, *pclose*() shall not return before the child process created by *popen*() has terminated.

If the command language interpreter cannot be executed, the child termination status returned by *pclose*() shall be as if the command language interpreter terminated using *exit*(127) or *_exit*(127). If it can be executed, the *exit*() value shall be as described for the `sh` utility.

The *pclose*() function shall not affect the termination status of any child of the calling process other than the one created by *popen*() for the associated stream.

If the argument *stream* to *pclose*() is not a pointer to a stream created by *popen*(), the result of *pclose*() is undefined.

After *popen*(), both the parent and the child process shall be capable of executing independently before either terminates. See 2.9.1.2.

B.3.2.3 Returns

The *popen*() function shall return a **NULL** pointer if the pipe or subprocess cannot be created. Otherwise, it shall return a stream pointer as described above.

Upon successful return, *pclose*() shall return the termination status of the command language interpreter. Otherwise, *pclose*() shall return −1 and set *errno* to indicate the error.

B.3.2.4 Errors

If any of the following conditions are detected, the *popen*() function shall return **NULL** and set *errno* to the corresponding value:

[EINVAL] The *mode* argument is invalid.

The *popen*() function may also set *errno* values as described by the POSIX.1 {8} *fork*() or *pipe*() functions.

If any of the following conditions are detected, the *pclose*() function shall return −1 and set *errno* to the corresponding value:

[ECHILD] The status of the child process could not be obtained, as described above.

B.3.2.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

Because open files are shared, a mode `"r"` command can be used as an input filter and a mode `"w"` command as an output filter.

The behavior of *popen*() is specified for *mode*s of `"r"` and `"w"`. Other modes such as `"rb"` and `"wb"` might be supported by specific implementations, but these would not be portable features. Note that historical implementations of *popen*() only check to see if the first character of *mode* is `r`. Thus, a *mode* of `"robert the robot"` would be treated as *mode* `"r"`, and a *mode* of `"anything else"` would be treated as *mode* `"w"`.

If the application calls *waitpid*() with a *pid* argument greater than zero, and it still has a *popen*()ed stream open, it must ensure that *pid* does not refer to the process started by *popen*().

History of Decisions Made

There is a requirement that *pclose*() not return before the child process terminates. This is intended to disallow implementations that return [EINTR] if a signal is received while waiting. If *pclose*() returned before the child terminated, there would be no way for the application to discover which child used to be associated with the stream, and it could not do the cleanup itself.

If the stream pointed to by *stream* was not created by *popen*(), historical implementations of *pclose*() return −1 without setting *errno*. To avoid requiring *pclose*() to set *errno* in this case, this standard makes the behavior undefined. An application should not use *pclose*() to close any stream that wasn't created by *popen*().

Wording was added in Draft 10 requiring that the parent and child processes be able to execute independently. This behavior has been the intent all along, and the specific words were taken from the current draft of the POSIX.1a revision to POSIX.1 {8}. Rationale about this wording appears in B.3.1.1 of POSIX.1a.

Some historical implementations either block or ignore the signals SIGINT, SIGQUIT, and SIGHUP while waiting for the child process to terminate. Since this behavior is not described in POSIX.2, such implementations are not conforming. Also, some historical implementations return [EINTR] if a signal is received, even though the child process has not terminated. Such implementations are also considered nonconforming.

Consider, for example, an application that uses

```
popen("command", "r")
```

to start *command*, which is part of the same application. The parent writes a prompt to its standard output (presumably the terminal) and then reads from the *popened* stream. The child reads the response from the user, does some transformation on the response (pathname expansion, perhaps) and writes the result to its standard output. The parent process reads the result from the pipe, does something with it, and prints another prompt. The cycle repeats. Assuming that both processes do appropriate buffer flushing, this would be expected to work.

Modified in Draft 6 to reflect the availability of the *waitpid*() function in POSIX.1 {8}. To conform to this standard, *pclose*() must use *waitpid*(), or some similar function, instead of *wait*().

Figure B-2 illustrates how the *pclose*() function might be implemented on a POSIX.1 {8} system.

```
int pclose(FILE *stream)                                          1
{
        int     stat;
        pid_t   pid;

        pid = <pid for process created for stream by popen()>
        (void) fclose(stream);

        while (waitpid(pid, &stat, 0) == -1) {
                if (errno != EINTR) {
                        stat = -1;
                        break;
                }
        }
        return(stat);
}
```

Figure B-2 – Sample *pclose*() Implementation

B.4 C Binding for Access Environment Variables

Function: *getenv* ()

The C language binding to the service described in 7.2 shall be the POSIX.1 {8} *getenv* () function.

B.5 C Binding for Regular Expression Matching

Functions: *regcomp* (), *regexec* (), *regfree* (), *regerror* ()

B.5.1 Synopsis

```
#include <sys/types.h>
#include <regex.h>

int regcomp(regex_t *preg, const char *pattern, int cflags);

int regexec(const regex_t *preg, const char *string,
        size_t nmatch, regmatch_t pmatch[], int eflags);

size_t regerror(int errcode, const regex_t *preg,
        char *errbuf, size_t errbuf_size);

void regfree(regex_t *preg);
```

B.5.2 Description

These functions shall interpret basic and extended regular expressions, as described in 2.8.

The header `<regex.h>` shall define the structure types *regex_t* and *regmatch_t*. The structure type *regex_t* shall include at least the member shown in Table B-6.

The structure type *regmatch_t* shall contain at least the members shown in
Table B-7. The type *regoff_t*, which shall be defined in `<regex.h>`, shall be a 1
signed arithmetic type that can hold the largest value that can be stored in either 1
an *off_t* or a *ssize_t*. 1

The *regcomp* () function shall compile the regular expression contained in the 1
string pointed to by the *pattern* argument and place the results in the structure 1
pointed to by *preg*. The *cflags* argument shall be the bitwise inclusive OR of zero
or more of the flags shown in Table B-8, which shall be defined in the header
`<regex.h>`.

The default regular expression type for *pattern* shall be a Basic Regular Expression. The application can specify Extended Regular Expressions using the REG_EXTENDED *cflags* flag.

If the function *regcomp* () succeeds, it shall return zero; otherwise it shall return nonzero, and the content of *preg* shall be undefined.

Table B-6 – Structure Type *regex_t*

Member Type	Member Name	Description
size_t	*re_nsub*	Number of parenthesized subexpressions.

Table B-7 – Structure Type *regmatch_t*

Member Type	Member Name	Description	
regoff_t	*rm_so*	Byte offset from start of *string* to start of substring.	1
regoff_t	*rm_eo*	Byte offset from start of *string* of the first character after the end of substring.	1 1

Table B-8 – *regcomp* () *cflags* Argument

flag	Description
REG_EXTENDED	Use Extended Regular Expressions.
REG_ICASE	Ignore case in match. See 2.8.2.
REG_NOSUB	Report only success/fail in *regexec* ().
REG_NEWLINE	Change the handling of <newline>, as described in the text.

Table B-9 – *regexec* () *eflags* Argument

flag	Description
REG_NOTBOL	The first character of the string pointed to by *string* is not the beginning of the line. Therefore, the circumflex character (^), when taken as a special character, shall not match the beginning of *string* .
REG_NOTEOL	The last character of the string pointed to by *string* is not the end of the line. Therefore, the dollar sign ($), when taken as a special character, shall not match the end of *string* .

If the REG_NOSUB flag was not set in *cflags* , then *regcomp* () shall set *re_nsub* to the number of parenthesized subexpressions [delimited by `\( \)` in basic regular expressions or `( )` in extended regular expressions] found in *pattern.*

The *regexec* () function shall compare the null-terminated string specified by *string* against the compiled regular expression *preg* initialized by a previous call to *regcomp* (). If it finds a match, *regexec* () shall return zero; otherwise it shall return nonzero indicating either no match or an error. The *eflags* argument shall be the bitwise inclusive OR of zero or more of the flags shown in Table B-9, which shall be defined in the header `<regex.h>`.

If *nmatch* is zero or REG_NOSUB was set in the *cflags* argument to *regcomp*(), then *regexec*() shall ignore the *pmatch* argument. Otherwise, the *pmatch* argument shall point to an array with at least *nmatch* elements, and *regexec*() shall fill in the elements of that array with offsets of the substrings of *string* that correspond to the parenthesized subexpressions of *pattern*: *pmatch*[*i*].*rm_so* shall be the byte offset of the beginning and *pmatch*[*i*].*rm_eo* shall be one greater than the byte offset of the end of substring *i*. (Subexpression *i* begins at the *i*th matched open parenthesis, counting from 1.) Offsets in *pmatch*[0] shall identify the substring that corresponds to the entire regular expression. Unused elements of *pmatch* up to *pmatch*[*nmatch* −1] shall be filled with −1. If there are more than *nmatch* subexpressions in *pattern* (*pattern* itself counts as a subexpression), then *regexec*() shall still do the match, but shall record only the first *nmatch* substrings.

When matching a basic or extended regular expression, any given parenthesized subexpression of *pattern* might participate in the match of several different substrings of *string*, or it might not match any substring even though the pattern as a whole did match. The following rules shall be used to determine which substrings to report in *pmatch* when matching regular expressions:

(1) If subexpression *i* in a regular expression is not contained within another subexpression, and it participated in the match several times, then the byte offsets in *pmatch*[*i*] shall delimit the last such match.

(2) If subexpression *i* is not contained within another subexpression, and it did not participate in an otherwise successful match, then the byte offsets in *pmatch*[*i*] shall be −1. A subexpression shall not participate in the match when:

   (a) * or `\{ \}` appears immediately after the subexpression in a basic regular expression, or *, ?, or `{ }` appears immediately after the subexpression in an extended regular expression, and the subexpression did not match (matched zero times), or

   (b) **|** is used in an extended regular expression to select this subexpression or another, and the other subexpression matched.

(3) If subexpression *i* is contained within another subexpression *j*, and *i* is not contained within any other subexpression that is contained within *j*, and a match of subexpression *j* is reported in *pmatch*[*j*], then the match or nonmatch of subexpression *i* reported in *pmatch*[*i*] shall be as described in (1) and (2) above, but within the substring reported in *pmatch*[*j*] rather than the whole string.

(4) If subexpression *i* is contained in subexpression *j*, and the byte offsets in *pmatch*[*j*] are −1, then the byte offsets in *pmatch*[*i*] also shall be −1.

(5) If subexpression *i* matched a zero-length string, then both byte offsets in *pmatch*[*i*] shall be the byte offset of the character or null terminator immediately following the zero-length string.

If, when *regexec*() is called, the locale is different than when the regular expression was compiled, the result is undefined.

If REG_NEWLINE is not set in *cflags*, then a `<newline>` character in *pattern* or *string* shall be treated as an ordinary character. If REG_NEWLINE is set, then `<newline>` shall be treated as an ordinary character except as follows:

(1) A `<newline>` in *string* shall not be matched by a period outside of a bracket expression (see 2.8.3.1.3) or by any form of a nonmatching list (see 2.8.3.2).

(2) A circumflex (^) in *pattern*, when used to specify expression anchoring (see 2.8.4.4 and 2.8.4.6), shall match the zero-length string immediately after a `<newline>` in *string*, regardless of the setting of REG_NOTBOL.

(3) A dollar-sign ($) in *pattern*, when used to specify expression anchoring, shall match the zero-length string immediately before a `<newline>` in *string*, regardless of the setting of REG_NOTEOL.

The *regfree*() function shall free any memory allocated by *regcomp*() associated with *preg*.

The *regerror*() function provides a mapping from error codes returned by *regcomp*() and *regexec*() to unspecified printable strings. It shall generate a string corresponding to the value of the *errcode* argument, which shall be the last nonzero value returned by *regcomp*() or *regexec*() with the given value of *preg*. If *errcode* is not such a value, the content of the generated string is unspecified. If *preg* is (*regexec_t*)0, but *errcode* is a value returned by a previous call to *regexec*() or *regcomp*(), then *regerror*() still shall generate an error string corresponding to the value of *errcode*, but it might not be as detailed under some implementations.

If the *errbuf_size* argument is not zero, *regerror*() shall place the generated string into the *errbuf_size*-byte buffer pointed to by *errbuf*. If the string (including the terminating null) cannot fit in the buffer, *regerror*() shall truncate the string and null-terminate the result.

If *errbuf_size* is zero, *regerror*() shall ignore the *errbuf* argument, but shall return the integer value described below.

If the *preg* argument to *regexec*() or *regfree*() is not a compiled regular expression returned by *regcomp*(), the result is undefined. A *preg* shall no longer be treated as a compiled regular expression after it is given to *regfree*().

B.5.3 Returns

On successful completion, the *regcomp*() function shall return zero. On successful completion, the *regexec*() function shall return zero to indicate that *string* matched *pattern*, or REG_NOMATCH (which shall be defined in `<regex.h>`) to indicate no match.

The *regerror*() function shall return the size of the buffer needed to hold the entire generated string, including the null termination. If the return value is greater than *errbuf_size*, the string returned in the buffer pointed to by *errbuf* has been truncated.

Table B-10 – *regcomp* (), *regexec* () Return Values

Error Code	Description
REG_NOMATCH	*regexec* () failed to match
REG_BADPAT	Invalid regular expression
REG_ECOLLATE	Invalid collating element referenced
REG_ECTYPE	Invalid character class type referenced
REG_EESCAPE	Trailing \ in pattern
REG_ESUBREG	Number in \ *digit* invalid or in error
REG_EBRACK	[ ] imbalance
REG_EPAREN	\(\) or ( ) imbalance
REG_EBRACE	\{ \} imbalance
REG_BADBR	Content of \{ \} invalid: Not a number, number too large, more than two numbers, first larger than second
REG_ERANGE	Invalid endpoint in range expression
REG_ESPACE	Out of memory
REG_BADRPT	?, *, or + not preceded by valid regular expression

B.5.4 Errors

If *regcomp* () or *regexec* () fails, it shall return a nonzero value indicating the type of failure. Table B-10 contains the names of macros for error codes that may be returned. If a code is returned, the interpretation shall be as given in the table. The implementation shall define the macros in Table B-10 in `<regex.h>`, and may define additional macros beginning with ‘REG_’’for other error codes.

If *regcomp* () detects an illegal regular expression, it may return REG_BADPAT, or it may return one of the error codes that more precisely describes the error.

B.5.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

An example of using the functions is shown in Figure B-3

The following demonstrates how the REG_NOTBOL flag could be used with *regexec* () to find all substrings in a line that match a pattern supplied by a user. (For simplicity of the example, very little error checking is done.)

```
(void) regcomp (&re, pattern, 0);
/* this call to regexec() finds the first match on the line */
error = regexec (&re, &buffer[0], 1, &pm, 0);
while (error == 0) {    /* while matches found */
        <substring found between pm.rm_sp and pm.rm_ep>
        /* This call to regexec() finds the next match */
        error = regexec (&re, pm.rm_ep, 1, &pm, REG_NOTBOL);
}
```

```
#include <regex.h>

/*
 * Match string against the extended regular expression in
 * pattern, treating errors as no match.
 *
 * Return 1 for match, 0 for no match.
 */

int
match(const char *string, const char *pattern)
{
        int     status;
        regex_t re;

        if (regcomp(&re, pattern, REG_EXTENDED|REG_NOSUB) != 0) {
                return(0);      /* report error */
        }
        status = regexec(&re, string, (size_t) 0, NULL, 0);
        regfree(&re);
        if (status != 0) {
                return(0);      /* report error */
        }
        return status == 0;
}
```

Figure B-3 – Example Regular Expression Matching

An application could use `regerror(code,preg,NULL,(size_t)0)` to find out how big a buffer is needed for the generated string, *malloc* () a buffer to hold the string, and then call *regerror* () again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most strings (perhaps 128 bytes), and then *malloc* () a larger buffer if it finds that this is too small.

The *regmatch* () function must fill in all *nmatch* elements of *pmatch* , where *nmatch* and *pmatch* are supplied by the application, even if some elements of *pmatch* do not correspond to subexpressions in *pattern* . The application writer should note that there is probably no reason for using a value of *nmatch* that is larger than *preg->re_nsub* .

History of Decisions Made

The REG_ICASE flag supports the operations taken by the `grep -i` option and the historical implementations of `ex` and `vi`. Including this flag will make it easier for application code to be written that does the same thing as these utilities.

The substrings reported in *pmatch* [] are defined using offsets from the start of the string rather than pointers. Since this is a new interface, there should be no impact on historical implementations or applications, and offsets should be just as easy to use as pointers. The change to offsets was made to facilitate future extensions in which the string to be searched is presented to *regexec* () in blocks, allowing a string to be searched that is not all in memory at once.

A new type *regoff_t* is used for the elements of *pmatch* [] to ensure that the appli- 1
cation can represent either the largest possible array in memory (important for a 1
POSIX.2-conforming application) or the largest possible file (important for an 1
application using the extension where a file is searched in chunks). 1

The working group has rejected, at least for now, the inclusion of a *regsub* () function that would be used to do substitutions for a matched regular expression. While such a routine would be useful to some applications, its utility would be much more limited than the matching function described here. Both regular expression parsing and substitution are possible to implement without support other than that required by the C Standard {7}, but matching is much more complex than substituting. The only ''difficult'' part of substitution, given the information supplied by *regexec* (), is finding the next character in a string when there can be multibyte characters. That is a much wider issue, and one that needs a more general solution.

The *errno* variable has not been used for error returns to avoid cluttering up the *errno* namespace for this feature.

In Draft 9, the interface was modified so that the matched substrings *rm_sp* and *rm_ep* are in a separate *regmatch_t* structure instead of in *regex_t*. This allows a single compiled regular expression to be used simultaneously in several contexts; in *main* () and a signal handler, perhaps, or in multiple threads of lightweight processes. (The *preg* argument to *regexec* () is declared with type `const`, so the implementation is not permitted to use the structure to store intermediate results.) It also allows an application to request an arbitrary number of substrings from a regular expression. (Previous versions reported only ten substrings.) The number of subexpressions in the regular expression is reported in *re_nsub* in *preg*. With this change to *regexec* (), consideration was given to dropping the REG_NOSUB flag, since the user can now specify this with a zero *nmatch* argument to *regexec* (). However, keeping REG_NOSUB allows an implementation to use a different (perhaps more efficient) algorithm if it knows in *regcomp* () that no subexpressions need be reported. The implementation is only required to fill in *pmatch* if *nmatch* is not zero and if REG_NOSUB is not specified. Note that the *size_t* type, as defined in the C Standard {7}, is unsigned, so the description of *regexec* () does not need to address negative values of *nmatch*.

The rules for reporting substrings of extended regular expressions are consistent with those used by Henry Spencer's ''almost public domain'' version of *regexec* ().

The REG_NOTBOL and REG_NOTEOL flags were added to *regexec* () in Draft 9. REG_NOTBOL was added to allow an application to do repeated searches for the same pattern in a line. If the pattern contains a circumflex character that should match the beginning of a line, then the pattern should only match when matched against the beginning of the line. Without the REG_NOTBOL flag, the application could rewrite the expression for subsequent matches, but in the general case this would require parsing the expression. The need for REG_NOTEOL is not as clear;

it was added for symmetry.

The addition of the *regerror* () function addresses the historical need for portable application programs to have access to error information more than ‘‘Function failed to compile/match your regular expression for unknown reasons.’’

This interface provides for two different methods of dealing with error conditions. The specific error codes (REG_EBRACE, for example), defined in `<regex.h>`, allow an application to recover from an error if it is so able. Many applications, especially those that use patterns supplied by a user, will not try to deal with specific error cases, but will just use *regerror* () to obtain a human-readable error message to present to the user.

The *regerror* () function uses a scheme similar to *confstr* () to deal with the problem of allocating memory to hold the generated string. The scheme used by *strerror* () in the C Standard {7} was considered unacceptable since it creates difficulties for multithreaded applications. (POSIX.4a, a standard for threads, started balloting in January 1991.) A different scheme used by *regerror* () in one draft of this standard was eliminated to improve internal consistency, and because the current interface produced greater consensus than the other.

The *preg* argument is provided to *regerror* () to allow an implementation to generate a more descriptive message than would be possible with *errcode* alone. An implementation might, for example, save the character offset of the offending character of the pattern in a field of *preg*, and then include that in the generated message string. The implementation may also ignore *preg*.

A REG_FILENAME flag was considered, but omitted. This flag caused *regexec* () to match patterns as described in 3.13 instead of regular expressions. This service is now provided by the *fnmatch* () function [see B.6].

B.6 C Binding for Match Filename or Pathname

Function: *fnmatch* ()

B.6.1 Synopsis

```
#include <fnmatch.h>
int fnmatch(const char *pattern, const char *string, int flags);
```

B.6.2 Description

The *fnmatch* () function shall match patterns as described in 3.13.1 and 3.13.2. It checks the string specified by the *string* argument to see if it matches the pattern specified by the *pattern* argument.

The *flags* argument modifies the interpretation of *pattern* and *string*. It is the bitwise inclusive OR of zero or more of the flags shown in Table B-11, which are defined in the header `<fnmatch.h>`. If the FNM_PATHNAME flag is set in *flags*, then a slash character in *string* shall be explicitly matched by a slash in *pattern*; it shall not be matched by either the asterisk or question-mark special characters, nor by a bracket expression. If the FNM_PATHNAME flag is not set, the slash character shall be treated as an ordinary character.

Table B-11 – *fnmatch* () *flags* Argument

flags	Description	
FNM_NOESCAPE	Disable backslash escaping	1
FNM_PATHNAME	Slash in *string* only matches slash in *pattern*	
FNM_PERIOD	Leading period in *string* must be exactly matched by period in *pattern*	

If FNM_NOESCAPE is not set in *flags*, a backslash character (\) in *pattern* fol- 1
lowed by any other character shall match that second character in *string*. In par-
ticular, '\\' shall match a backslash in *string*. If FNM_NOESCAPE is set, a 1
backslash character shall be treated as an ordinary character.

If FNM_PERIOD is set in *flags*, then a leading period in *string* shall match a period
in *pattern* as described by rule (2) in 3.13.2, where the location of ''leading'' is indi- 1
cated by the value of FNM_PATHNAME: 1

— If FNM_PATHNAME is set, a period is ''leading'' if it is the first character in *string* or if it immediately follows a slash.

— If FNM_PATHNAME is not set, a period is ''leading'' only if it is the first character of *string*.

If FNM_PERIOD is not set, then no special restrictions shall be placed on matching a period.

B.6.3 Returns

If *string* matches the pattern specified by *pattern*, then *fnmatch*() shall return zero. If there is no match, *fnmatch*() shall return FNM_NOMATCH, which shall be defined in the header `<fnmatch.h>`. If an error occurs, *fnmatch*() shall return another nonzero value.

B.6.4 Errors

This standard does not specify any error conditions that are required to be detected by the *fnmatch*() function. Some errors may be detected under unspecified conditions.

B.6.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The *fnmatch*() function has two major uses. It could be used by an application or utility that needs to read a directory and apply a pattern against each entry. The `find` utility is an example of this. It can also be used by the `pax` utility to process its *pattern* operands, or by applications that need to match strings in a similar manner.

History of Decisions Made

This function replaces the REG_FILENAME flag of *regcomp*() in early drafts. It provides virtually the same functionality as the *regcomp*() and *regexec*() functions using the REG_FILENAME and REG_FSLASH flags [the REG_FSLASH flag was proposed for *regcomp*(), and would have had the opposite effect from FMN_PATHNAME], but with a simpler interface and less overhead.

The name *fnmatch*() is intended to imply *filename* match, rather than *pathname* match. The default action of this function is to match filenames, rather than pathnames, since it gives no special significance to the slash character. With the FNM_PATHNAME flag, *fnmatch*() does match pathnames, but without tilde expansion, parameter expansion, or special treatment for period at the beginning of a filename.

B.7 C Binding for Command Option Parsing

Function: *getopt* ()

B.7.1 Synopsis

```
#include <unistd.h>

int getopt(int argc, char * const argv[], const char *optstring);   1

extern char *optarg;
extern int optind, opterr, optopt;
```

B.7.2 Description

The *getopt* () function is a command-line parser that can be used by applications that follow Utility Syntax Guidelines 3, 4, 5, 6, 7, 9, and 10 in 2.10.2. The remaining guidelines are not addressed by *getopt* () and are the responsibility of the application.

The parameters *argc* and *argv* are the argument count and argument array as passed to *main* (). The argument *optstring* is a string of recognized option characters; if a character is followed by a colon, the option takes an argument. All option characters allowed by Utility Syntax Guideline 3 are allowed in *optstring* . The implementation may accept other characters as an extension.

The variable *optind* is the index of the next element of the *argv* [] vector to be processed. It is initialized to 1 by the system, and *getopt* () updates it when it finishes with each element of *argv* []. When an element of *argv* [] contains multiple option characters, it is unspecified how *getopt* () determines which options have already been processed.

The *getopt* () function shall return the next option character from *argv* that matches a character in *optstring* , if there is one that matches. If the option takes 1
an argument, *getopt* () shall set the variable *optarg* to point to the option-argument as follows:

(1) If the option was the last character in the string pointed to by an element of *argv* , then *optarg* contains the next element of *argv* , and *optind* shall be incremented by 2. If the resulting value of *optind* is not less than *argc* , this indicates a missing option argument, and *getopt* () shall return an error indication.

(2) Otherwise, *optarg* points to the string following the option character in that element of *argv* , and *optind* shall be incremented by 1.

If, when *getopt* () is called, *argv* [*optind*] is **NULL** , **argv* [*optind*] is not the character –, or *argv* [*optind*] points to the string `"-"`, *getopt* () shall return −1 without changing *optind* . If *argv* [*optind*] points to the string `"--"`, *getopt* () shall return −1 after incrementing *optind* .

If *getopt* () encounters an option character that is not contained in *optstring* , it shall return the question-mark (`?`) character. If it detects a missing option argument, it shall return the colon character (`:`) if the first character of *optstring* was a colon, or a question-mark character otherwise. In either case, *getopt* () shall set the variable *optopt* to the option character that caused the error. If the application has not set the variable *opterr* to zero and the first character of *optstring* is not a colon, *getopt* () shall also print a diagnostic message to standard error using the formatting rules specified for the `getopts` utility (see 4.27.6.2).

B.7.3 Returns

The *getopt* () function shall return the next option character specified on the command line. The value −1 shall be returned when all command line options have been parsed.

B.7.4 Errors

If an invalid option is encountered, *getopt* () shall return a question-mark character. If an option with a missing option argument is encountered, *getopt* () shall return either a question-mark or a colon, as described previously.

B.7.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The *getopt* () function is only required to support option characters included in Guideline 3. Many historical implementations of *getopt* () support other characters as options. This is an allowed extension, but applications that use extensions are not maximally portable. Note that support for multibyte option characters is only possible when such characters can be represented as type *int* .

The code fragment in Figure B-4 shows how one might process the arguments for a utility that can take the mutually exclusive options `a` and `b` and the options `f` and `o`, both of which require arguments.

The code in Figure B-4 accepts any of the following as equivalent:

```
cmd -ao arg path path
cmd -a -o arg path path
cmd -o arg -a path path
cmd -a -o arg -- path path
cmd -a -oarg path path
cmd -aoarg path path
```

History of Decisions Made

Support for the *optopt* variable was added in Draft 9. This documents historical practice, and allows the application to obtain the identity of the invalid option.

```
#include  <unistd.h>
int main (int argc, char *argv[ ])
{
        int c, bflg, aflg, errflg = 0;
        char *ifile, *ofile;
        extern char *optarg;
        extern int optind, optopt;
        . . .
        while ((c = getopt(argc, argv, ":abf:o:")) != -1) {
                switch (c) {
                case 'a':
                        if (bflg)
                                errflg = 1;
                        else
                                aflg = 1;
                        break;
                case 'b':
                        if (aflg)
                                errflg = 1;
                        else
                                bflg = 1;
                                bproc( );
                        break;
                case 'f':
                        ifile = optarg;
                        break;
                case 'o':
                        ofile = optarg;
                        break;
                case ':':       /* -f or -o without option-arg */
                        fprintf (stderr,
                                "Option -%c requires an option-argument\n",
                                optopt);
                        errflg = 1;
                        break;
                case '?':
                        fprintf (stderr,
                                "Unrecognized option: -%c\n", optopt);
                        errflg = 1;
                        break;
                }
        }
        if (errflg) {
                fprintf(stderr, "usage: . . . ");
                exit(2);
        }
        for ( ; optind < argc; optind++) {
                if (access(argv[optind], R_OK)) {
        . . .
}
```

Figure B-4 – Argument Processing with *getopt* ()

The description was extensively rewritten in Draft 9 to be more explicit about how *optarg* and *optind* are set, and to recognize that this routine deals with a vector of string pointers, not directly with a shell command line.

The description was modified in Draft 9 to make it clear that *getopt* (), like the `getopts` utility, shall deal with option-arguments whether separated from the option by `<blank>`s or not. Note that the requirements on *getopt* () and `getopts` are more stringent than the Utility Syntax Guidelines.

The *getopt* () function has been changed to return `−1`, rather than `EOF`, so that *<stdio.h>* is not required.

The special significance of a colon as the first character of *optstring* was added in Draft 11 to make *getopt* () consistent with the `getopts` utility. It allows an application to make a distinction between a missing argument and an incorrect option letter without having to examine the option letter. It is true that a missing argument can only be detected in one case, but that is a case that has to be considered.

B.8 C Binding for Generate Pathnames Matching a Pattern

Functions: *glob* (), *globfree* ()

B.8.1 Synopsis

```
#include <glob.h>

int glob(const char *pattern , int flags ,
        int (*errfunc )(const char *epath , int eerrno ), glob_t *pglob);

void globfree(glob_t *pglob);
```

B.8.2 Description

The *glob* () function is a pathname generator that implements the rules defined in 3.13, with optional support for rule (3) in 3.13.3.

The header `<glob.h>` defines the structure type *glob_t*, which includes at least the members shown in Table B-12.

The argument *pattern* is a pointer to a pathname pattern to be expanded. The *glob* () function shall match all accessible pathnames against this pattern and develop a list of all pathnames that match. In order to have access to a pathname, *glob* () requires search permission on every component of a path except the last and read permission on each directory of any filename component of *pattern* that contains any of the special characters `*`, `?` or `[`. The *glob* () function stores the number of matched pathnames into *pglob->gl_pathc* and a pointer to a list of pointers to pathnames into *pglob->gl_pathv* . The pathnames are in sort order as defined by 2.2.2.30. The first pointer after the last pathname shall be **NULL** . If the pattern does not match any pathnames, the returned number of matched paths is set to zero.

Table B-12 – Structure Type *glob_t*

Member Type	Member Name	Description	
size_t	*gl_pathc*	Count of paths matched by *pattern*.	1
char **	*gl_pathv*	Pointer to a list of matched pathnames.	
size_t	*gl_offs*	Slots to reserve at the beginning of *gl_pathv*.	1

It is the caller's responsibility to create the structure pointed to by *pglob*. The *glob*() function shall allocate other space as needed, including the memory pointed to by *gl_pathv*. The *globfree*() function shall free any space associated with *pglob* from a previous call to *glob*().

The argument *flags* is used to control the behavior of *glob*(). The value of *flags* is the bitwise inclusive OR of any of the constants shown in Table B-13, which are defined in `<glob.h>`.

Table B-13 – *glob*() *flags* Argument

Name	Description	
GLOB_APPEND	Append pathnames generated to the ones from a previous call to *glob*().	
GLOB_DOOFFS	Make use of *pglob->gl_offs*. If this flag is set, *pglob->gl_offs* is used to specify how many **NULL** pointers to add to the beginning of *pglob->gl_pathv*. In other words, *pglob->gl_pathv* shall point to *pglob->gl_offs* **NULL** pointers, followed by *pglob->gl_pathc* pathname pointers, followed by a **NULL** pointer.	
GLOB_ERR	Causes *glob*() to return when it encounters a directory that it cannot open or read. Ordinarily, *glob*() continues to find matches.	
GLOB_MARK	Each pathname that is a directory that matches *pattern* has a slash appended.	
GLOB_NOCHECK	Support rule (3) in 3.13.3. If *pattern* does not match any pathname, then *glob*() shall return a list consisting of only *pattern*, and the number of matched pathnames is 1.	
GLOB_NOESCAPE	Disable backslash escaping.	1
GLOB_NOSORT	Ordinarily, *glob*() sorts the matching pathnames according to the definition of *collation sequence* in 2.2.2.30. When this flag is used the order of pathnames returned is unspecified.	

The GLOB_APPEND flag can be used to append a new set of words to those gen-
erated by a previous call to *glob*(). The following rules apply when two or more 1
calls to *glob*() are made with the same value of *pglob* and without intervening 1
calls to *globfree*(): 1

(1) The first such call shall not set GLOB_APPEND. All subsequent calls 1
shall set it. 1

(2) All of the calls shall set GLOB_DOOFFS, or all shall not set it. 1

(3) After the second call, *pglob->gl_pathv* shall point to a list containing the following:

(a) Zero or more **NULL**s, as specified by GLOB_DOOFFS and *pglob->gl_offs*.

(b) Pointers to the pathnames that were in the *pglob->gl_pathv* list before the call, in the same order as before.

(c) Pointers to the new pathnames generated by the second call, in the specified order.

(4) The count returned in *pglob->gl_pathc* shall be the total number of pathnames from the two calls.

The application can change any of the fields in Table B-12 after a call to *glob*(), 1
but if it does it shall reset them to the original value before a subsequent call, 1
using the same *pglob* value, to *globfree*() or *glob*() with the GLOB_APPEND flag. 1

If, during the search, a directory is encountered that cannot be opened or read and *errfunc* is not **NULL**, *glob*() shall call (**errfunc*)() with two arguments:

(1) The *epath* argument is a pointer to the path that failed.

(2) The *eerrno* argument is the value of *errno* from the failure, as set by the POSIX.1 {8} *opendir*(), *readdir*(), or *stat*() functions. (Other values may be used to report other errors not explicitly documented for those functions.)

If (**errfunc*)() is called and returns nonzero, or if the GLOB_ERR flag is set in *flags*, *glob*() shall stop the scan and return GLOB_ABORTED after setting *gl_pathc* and *gl_pathv* in *pglob* to reflect the paths already scanned. If GLOB_ERR is not set and either *errfunc* is **NULL** or (**errfunc*)() returns zero, the error shall be ignored.

B.8.3 Returns

On successful completion, *glob*() shall return zero. The argument *pglob->gl_pathc* shall return the number of matched pathnames and the argument *pglob->gl_pathv* shall contain a pointer to a null-terminated list of matched and sorted pathnames. However, if *pglob->gl_pathc* is zero, the content of *pglob->gl_pathv* is undefined.

B.8.4 Errors

If *glob*() terminates due to an error, it shall return one of the nonzero constants shown in Table B-14, which are defined in <glob.h>. The arguments *pglob->gl_pathc* and *pglob->gl_pathv* are still set as defined above in Returns.

Table B-14 – *glob* () Error Return Values

Name	Description	
GLOB_ABORTED	The scan was stopped because GLOB_ERR was set or (*errfunc)() returned nonzero.	
GLOB_NOMATCH	The *pattern* does not match any exiting pathname, and GLOB_NOCHECK was not set in *flags*.	1
GLOB_NOSPACE	An attempt to allocate memory failed.	

B.8.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

This function is not provided for the purpose of enabling utilities to perform pathname expansion on their arguments, as this operation is performed by the shell, and utilities are explicitly not expected to redo this. Instead, it is provided for applications that need to do pathname expansion on strings obtained from other sources, such as a pattern typed by a user or read from a file.

If a utility needs to see if a pathname matches a given pattern, it can use *fnmatch* ().

Note that *gl_pathc* and *gl_pathv* have meaning even if *glob* () fails. This allows *glob* () to report partial results in the event of an error. However, if *gl_pathc* is zero, *gl_pathv* is unspecified even if *glob* () did not return an error.

The GLOB_NOCHECK option could be used when an application wants to expand a pathname if wildcards are specified, but wants to treat the pattern as just a string otherwise. The `sh` utility might use this for option-arguments, for example.

One use of the GLOB_DOOFFS flag is by applications that build an argument list for use with the POSIX.1 {8} *execv* (), *execve* (), or *execvp* () functions. Suppose, for example, that an application wants to do the equivalent of `ls -l *.c`, but for some reason `system("ls -l *.c")` is not acceptable. The application could obtain (*approximately*) the same result using the sequence:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
globbuf.gl_pathv[0] = "ls";
globbuf.gl_pathv[1] = "-l";
execvp ("ls", &globbuf.gl_pathv[0]);
```

Using the same example, `ls -l *.c *.h` could be approximately simulated using GLOB_APPEND as follows:

```
globbuf.gl_offs = 2;
glob ("*.c", GLOB_DOOFFS, NULL, &globbuf);
glob ("*.h", GLOB_DOOFFS|GLOB_APPEND, NULL, &globbuf);
 ... etc. ...
```

The new pathnames generated by a subsequent call with GLOB_APPEND are not sorted together with the previous pathnames. This mirrors the way that the shell handles pathname expansion when multiple expansions are done on a command line.

History of Decisions Made

The interface was simplified to a useful, but less complex, subset. The *errfunc* argument was added to allow errors to be reported.

A reviewer claimed that the GLOB_DOOFFS flag is unnecessary because it could be simulated using:

```
new = (char **)malloc((n + pglob->gl_pathc + 1)
               * sizeof (char *));
(void) memcpy (new+n, pglob->gl_pathv,
               pglob->gl_pathc * sizeof(char *));
(void) memset (new, 0, n * sizeof (char *));
free (pglob->gl_pathv);
pglob->gl_pathv = new;
```

However, this assumes that the memory pointed to by *gl_pathv* is a block that was separately created using *malloc* (). This is not necessarily the case. An application should make no assumptions about how the memory referenced by fields in *pglob* was allocated. It might have been obtained from *malloc* () in a large chunk, and then carved up within *glob* (), or it might have been created using a different memory allocator. It is not the intent of this standard to specify or imply how the memory used by *glob* () is managed.

The structure elements *gl_pathc* and *gl_pathv* were renamed from *gl_argc* and *gl_argv* in Draft 9. The old names implied an association with the parameters to *main* () that does not necessarily exist.

The GLOB_APPEND flag was added in Draft 9 at the request of a reviewer. This flag would be used when an application wants to expand several different patterns into a single list.

Tilde and parameter expansion were removed from *glob* () in Draft 9. Applications that need these expansions should use the *wordexp* () function [see B.9].

B.9 C Binding for Perform Word Expansions

Functions: *wordexp* (), *wordfree* ()

B.9.1 Synopsis

```
#include <wordexp.h>
int wordexp(const char *words, wordexp_t *pwordexp, int flags);
void wordfree(wordexp_t *pwordexp);
```

B.9.2 Description

The *wordexp* () function shall perform word expansions as described in 3.6, subject to quoting as in 3.2, and place the list of expanded words into *pwordexp*. The expansions shall be the same as would be performed by the shell if *words* were the part of a command line representing the arguments to a utility. Therefore, *words* shall not contain an unquoted `<newline>` or any of the unquoted shell special characters |, &, ;, <, or >, except in the context of command substitution as specified in 3.6.3. It also shall not contain unquoted parentheses or braces, except in the context of command or variable substitution. If *words* contains an unquoted comment character (number sign) that is the beginning of a token, *wordexp* () may treat the comment character as a regular character, or may interpret it as a comment indicator and ignore the remainder of *words*.

The header `<wordexp.h>` defines the structure type *wordexp_t*, which includes at least the members shown in Table B-15.

Table B-15 – Structure Type *wordexp_t*

Member Type	Member Name	Description	
size_t	*we_wordc*	Count of words matched by *words*.	1
char **	*we_wordv*	Pointer to list of expanded words.	
size_t	*we_offs*	Slots to reserve at the beginning of *we_wordv*.	1

The argument *words* is a pointer to a string containing one or more words to be expanded. The *wordexp* () function shall store the number of generated words into *we_wordc* and a pointer to a list of pointers to words in *we_wordv*. Each individual field created during field splitting (see 3.6.5) or pathname expansion (see 3.6.6) is a separate word in the *we_wordv* list. The words are in order as described in 3.6. The first pointer after the last word pointer shall be **NULL**. The expansion of special parameters described in 3.5.2 is unspecified.

It is the caller's responsibility to create the structure pointed to by *pwordexp*. The *wordexp* () function allocates other space as needed, including memory pointed to by *we_wordv* . The *wordfree* () function shall free any memory associated with *pwordexp* from a previous call to *wordexp* ().

The argument *flags* is used to control the behavior of *wordexp* (). The value of *flags* is the bitwise inclusive OR of any of the constants in Table B-16, which are defined in <wordexp.h>.

Table B-16 – *wordexp* () *flags* Argument

Name	Description
WRDE_APPEND	Append words generated to the ones from a previous call to *wordexp* ().
WRDE_DOOFFS	Make use of *we_offs* . If this flag is set, *we_offs* is used to specify how many **NULL** pointers to add to the beginning of *we_wordv* . In other words, *we_wordv* shall point to *we_offs* **NULL** pointers, followed by *we_wordc* word pointers, followed by a **NULL** pointer.
WRDE_NOCMD	Fail if command substitution, as specified in 3.6.3, is requested.
WRDE_REUSE	The *pwordexp* argument was passed to a previous successful call to *wordexp* (), and has not been passed to *wordfree* (). The result shall be the same as if the application had called *wordfree* () and then called *wordexp* () without WRDE_REUSE.
WRDE_SHOWERR	Do not redirect standard error to `/dev/null`.
WRDE_UNDEF	Report error on an attempt to expand an undefined shell variable.

The WRDE_APPEND flag can be used to append a new set of words to those generated by a previous call to *wordexp* (). The following rules apply when two or more calls to *wordexp* () are made with the same value of *pwordexp* and without intervening calls to *wordfree* ():

(1) The first such call shall not set WRDE_APPEND. All subsequent calls shall set it.

(2) All of the calls shall set WRDE_DOOFFS , or all shall not set it.

(3) After the second and each subsequent call, *we_wordv* shall point to a list containing the following:

(a) Zero or more **NULL** s, as specified by WRDE_DOOFFS and *we_offs* .

(b) Pointers to the words that were in the *we_wordv* list before the call, in the same order as before.

(c) Pointers to the new words generated by the latest call, in the specified order.

(4) The count returned in *we_wordc* shall be the total number of words from all of the calls.

The application can change any of the fields in Table B-15 after a call to *wor-* 1
dexp (), but if it does it shall reset them to the original value before a subsequent 1
call, using the same *pwordexp* value, to *wordfree* () or *wordexp* () with the 1
WRDE_APPEND or WRDE_REUSE flag. 1

If *words* contains an unquoted `<newline>`, |, &, `;`, <, >, parenthesis, or brace in an inappropriate context, *wordexp* () shall fail, and the number of expanded words shall be zero.

Unless WRDE_SHOWERR is set in *flags*, *wordexp* () shall redirect standard error to `/dev/null` for any utilities executed as a result of command substitution while expanding *words*. If WRDE_SHOWERR is set, *wordexp* () may write messages to standard error if syntax errors are detected while expanding *words*.

If WRDE_DOOFFS is set, then *we_offs* shall have the same value for each *wor-* 1
dexp () call and the *wordfree* () call using a given *pglob*. 1

B.9.3 Returns

If no errors are encountered while expanding *words*, *wordexp* () shall return zero. Otherwise it shall return a nonzero value.

B.9.4 Errors

Table B-17 – *wordexp* () Return Values

Name	Description
WRDE_BADCHAR	One of the unquoted characters |, &, ;, <, >, parentheses, or braces appears in *words* in an inappropriate context.
WRDE_BADVAL	Reference to undefined shell variable when WRDE_UNDEF is set in *flags*.
WRDE_CMDSUB	Command substitution requested when WRDE_NOCMD was set in flags.
WRDE_NOSPACE	Attempt to allocate memory failed
WRDE_SYNTAX	Shell syntax error, such as unbalanced parentheses or unterminated string.

If *wordexp* () terminates due to an error, it shall return one of the nonzero constants shown in Table B-17, which shall be defined in `<wordexp.h>`. The implementation may define additional error returns beginning with WRDE_.

If *wordexp* () returns the error value WRDE_NOSPACE, then *pwordexp->we_wordc* and *pwordexp->we_wordv* shall be updated to reflect any words that were successfully expanded. In other cases, they shall not be modified.

B.9.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

This function is intended to be used by an application that wants to do all of the shell's expansions on a word or words obtained from a user. For example, if the application prompts for a file name (or list of file names) and then used *wordexp* () to process the input, the user could respond with anything that would be valid as input to the shell.

The WRDE_NOCMD flag is provided for applications that, for security or other reasons, want to prevent a user from executing shell commands. Disallowing unquoted shell special characters also prevents unwanted side effects such as executing a command or writing a file.

History of Decisions Made

This function was added in Draft 9 as an alternative to *glob* (). There has been continuing controversy over exactly what features should be included in *glob* (). It is hoped that providing *wordexp* () (which provides all of the shell's word expansions, but will probably be slow to execute), and *glob* () (which is faster but does only expansion of pathnames, without tilde or parameter expansion), will satisfy the majority of reviewers.

While *wordexp* () could be implemented entirely as a library routine, it is expected that most implementations will run a shell in a subprocess to do the expansion.

Two different approaches have been proposed for how the required information might be presented to the shell and the results returned. They are presented here as examples.

One proposal is to extend the `echo` utility by adding a `-q` option. This option would cause `echo` to add a backslash before each backslash and each `<blank>` that occurs within an argument. The *wordexp* () function could then invoke the shell as follows:

```
(void) strcpy (buffer, "echo -q ");
(void) strcat (buffer, words );
if ((flags & WRDE_SHOWERR) == 0)
        (void) strcat (buffer, " 2>/dev/null");
f = popen (buffer, "r");
```

The *wordexp* () function would read the resulting output, remove unquoted backslashes, and break into words at unquoted `<blank>`s. If the WRDE_NOCMD flag was set, *wordexp* () would have to scan *words* before starting the subshell to make sure that there would be no command substitution. In any case, it would have to scan *words* for unquoted special characters.

Another proposal is to add the following options to `sh`:

`-w` *wordlist* This option provides a wordlist expansion service to applications. The words in *wordlist* are expanded, and the following is written to standard output:

(1) The count of the number of words after expansion, in decimal, followed by a null byte.

(2) The number of bytes needed to represent the expanded words (not including null separators), in decimal, followed by a null byte.

(3) The expanded words, each terminated by a null byte.

If an error is encountered during word expansion, `sh` exits with a nonzero status after writing the above to report any words successfully expanded

`-P` Run in ‘‘protected’’ mode. If specified with the `-w` option, no command substitution is performed.

With these options, *wordexp* () could be implemented fairly simply by creating a subprocess using *fork* (), and executing `sh` using the line:

```
execl(<shell path>, "sh", "-P", "-w", words, (char *)0);
```

after directing standard error to `/dev/null`.

It seemed objectionable for a library routine to write messages to standard error, unless explicitly requested, so *wordexp* () is required to redirect standard error to `/dev/null` to ensure that no messages are generated, even for commands executed for command substitution. The new WRDE_SHOWERR flag can be specified to request that error messages be written.

The WRDE_REUSE flag allows the implementation to avoid the expense of freeing and reallocating memory, if that is possible. A minimal implementation can just call *wordfree* () when WRDE_REUSE is set.

B.10 C Binding for Get POSIX Configurable Variables

B.10.1 C Binding for Get String-Valued Configurable Variables

Function: *confstr* ()

B.10.1.1 Synopsis

```
#include <unistd.h>
size_t confstr(int name , char *buf, size_t len );
```

B.10.1.2 Description

The *confstr* () function provides a method for applications to get configuration-defined string values. Its use and purpose are similar to the *sysconf* () function defined in POSIX.1 {8}, but it is used where string values rather than numeric values are returned.

The *name* argument represents the system variable to be queried. The implementation shall support all of the *name* values shown in Table B-18, which are defined in `<unistd.h>`. It may support others.

Table B-18 – `confstr()` *name* Values

***name* Value**	**String returned by `confstr()`**
_CS_PATH	A value for the **PATH** environment variable that finds all standard utilities.

If *len* is not zero, and if *name* has a configuration-defined value, *confstr* () shall copy that value into the *len*-byte buffer pointed to by *buf*. If the string to be returned is longer than *len* bytes, including the terminating null, then *confstr* () shall truncate the string to $len-1$ bytes and null-terminate the result. The application can detect that the string was truncated by comparing the value returned by *confstr* () with *len*.

If *len* is zero and *buf* is **NULL**, then *confstr* () still shall return the integer value as defined below, but shall not return a string. If *len* is zero but *buf* is not **NULL**, the result is unspecified.

B.10.1.3 Returns

If *name* does not have a configuration-defined value, *confstr* () shall return zero and leave *errno* unchanged.

If *name* has a configuration-defined value, the *confstr* () function shall return the size of buffer that would be needed to hold the entire configuration-defined value.

If this return value is greater than *len*, the string returned in *buf* has been truncated.

B.10.1.4 Errors

If any of the following conditions occur, *confstr*() shall return zero and set *errno* to the corresponding value:

[EINVAL] The value of the *name* argument is invalid.

B.10.1.5 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

An application can distinguish between an invalid *name* parameter value and one that corresponds to a configurable variable that has no configuration-defined value by checking if *errno* has been modified. This mirrors the behavior of *sysconf*() in POSIX.1 {8}.

The original need for this function was to provide a way of finding the configuration-defined default value for the environment variable **PATH**. Since **PATH** can be modified by the user to include directories that could contain utilities replacing POSIX.2 standard utilities, applications need a way to determine the system-supplied **PATH** environment variable value that contains the correct search path for the POSIX.2 standard utilities.

An application could use `confstr(name,NULL,(size_t) 0)` to find out how big a buffer is needed for the string value, *malloc*() a buffer to hold the string, and call *confstr*() again to get the string. Alternately, it could allocate a fixed, static buffer that is big enough to hold most answers (512 bytes, maybe, or 1024), but then *malloc*() a larger buffer if it finds that this is too small.

History of Decisions Made

In Draft 7, these values and *sysconf*() values defined in POSIX.1 {8} were obtained using a function named *posixconf*(). However, that routine was dropped in favor of *csysconf*(). There did not seem to be any reason to provide the redundant interface to POSIX.1 {8} functions, nor to return values as strings when numeric values are really what are needed. *csysconf*() could be extended to return strings for other related standards or features.

In Draft 9, *csysconf*() has been replaced by *confstr*(). The name was changed because too many people were confused by the name; they thought that the 'c' referred to the C language, rather than characters (as distinct from integers). The *confstr*() function also copies the returned string into a buffer supplied by the application instead of returning a pointer to a string. This allows a cleaner interface in some implementations (lightweight processes were mentioned), and resolves questions about when the application must copy the string returned.

B.10.2 C Binding for Get Numeric-Valued Configurable Variables

Functions: *sysconf* (), *pathconf* (), *fpathconf* ()

A system that supports the C Language Bindings Option shall support the C language bindings defined in POSIX.1 {8} for the *sysconf* (), *pathconf* (), and *fpathconf* () functions. Of the *name* values defined in POSIX.1 {8}, only those that correspond to numeric-valued configuration values listed in Table 7-1, are required by POSIX.2. In addition, the *sysconf* () function shall support the *name* values in Table B-19, defined in `<unistd.h>`, to provide values for values in 2.13.1.

Table B-19 – C Bindings for Numeric-Valued Configurable Variables

Symbolic Limit	*name* Value
{BC_BASE_MAX}	_SC_BC_BASE_MAX
{BC_DIM_MAX}	_SC_BC_DIM_MAX
{BC_SCALE_MAX}	_SC_BC_SCALE_MAX
{BC_STRING_MAX}	_SC_BC_STRING_MAX
{COLL_WEIGHTS_MAX}	_SC_COLL_WEIGHTS_MAX
{EXPR_NEST_MAX}	_SC_EXPR_NEST_MAX
{LINE_MAX}	_SC_LINE_MAX
{RE_DUP_MAX}	_SC_RE_DUP_MAX
{POSIX2_VERSION}	_SC_2_VERSION
{POSIX2_C_DEV}	_SC_2_C_DEV
{POSIX2_FORT_DEV}	_SC_2_FORT_DEV
{POSIX2_FORT_RUN}	_SC_2_FORT_RUN
{POSIX2_LOCALEDEF}	_SC_2_LOCALEDEF
{POSIX2_SW_DEV}	_SC_2_SW_DEV

B.10.3 Rationale. *(This subclause is not a part of P1003.2)*

In Draft 9, the *name* values corresponding to the _POSIX2_* symbolic limits were changed to more closely follow the convention used in POSIX.1 {8}. In POSIX.1 {8}, for example, the *name* value for {_POSIX_VERSION} is _SC_VERSION. The POSIX.2 *name* value for {_POSIX2_C_DEV} (actually, it was {_POSIX_C_DEV} in Draft 8) was _SC_POSIX_C_DEV, and is now _SC_2_C_DEV.

If `sysconf(_SC_2_VERSION)` is not equal to the value of the {_POSIX2_VERSION} symbolic constant (see B.2.2), the utilities available via *system* () or *popen* () might not behave as described in this standard. This would mean that the application is not running in an environment that conforms to POSIX.2. Some applications might be able to deal with this, others might not. However, the interfaces defined in Annex B shall continue to operate as specified, even if `sysconf(_SC_2_VERSION)` reports that the utilities no longer perform as specified.

B.11 C Binding for Locale Control

The C binding to the services described in 7.9 shall be the *setlocale* () function defined in POSIX.1 {8} 8.1.2. In addition to the category values defined in POSIX.1 {8}, *setlocale* () shall also accept the value LC_MESSAGES , which shall be defined in `<locale.h>`.

B.11.1 C Binding for Locale Control Rationale. *(This subclause is not a part of P1003.2)*

The order in which the various locale categories are processed by *setlocale* () is not specified by POSIX.1 {8}, so the place for LC_MESSAGES in that order is also unspecified.

Annex C
(normative)

FORTRAN Development and Runtime Utilities Options

This annex describes utilities used for the development of FORTRAN language applications, including compilation or translation of FORTRAN source code, and the execution of certain FORTRAN applications at runtime.

The utilities described in this annex may be provided by the conforming system; however, any system claiming conformance to the FORTRAN Development Utilities Option shall provide the `fort77` utility and any system claiming conformance to the FORTRAN Runtime Utilities Option shall provide the `asa` utility.

C.0.1 FORTRAN Development and Runtime Utilities Options Rationale.

(This subclause is not a part of P1003.2)

This clause is included in this standard as a temporary measure to accommodate existing FORTRAN developers. It is the intention of the POSIX.2 working group that this annex be moved from this standard to the emerging standard being developed by the POSIX.9 working group, which will specify FORTRAN-specific interfaces to the basic services provided by this standard and POSIX.1. The movement of this annex should occur in a later version of this standard.

See the rationale for `asa` for a description of the FORTRAN Runtime Utilities Option and why it was split off from the FORTRAN Development Utilities Option.

C.1 asa — Interpret carriage-control characters

This utility is optional. It shall be provided on systems that support the FORTRAN Runtime Utilities Option.

C.1.1 Synopsis

`asa` [*file* ...]

C.1.2 Description

The `asa` utility shall write its input files to standard output, mapping carriage-control characters from the text files to line-printer control sequences in an implementation-defined manner.

The first character of every line shall be removed from the input, and the following actions shall be performed:

If the character removed is:

- `<space>` The rest of the line shall be output without change.
- `0` A `<newline>` shall be output, then the rest of the input line.
- `1` One or more implementation-defined characters that causes an advance to the next page shall be output, followed by the rest of the input line.
- `+` The `<newline>` of the previous line shall be replaced with one or more implementation-defined 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 shall have the same effect as `<space>`.

The action of the `asa` utility is unspecified upon encountering any character other than those listed above as the first character in a line.

C.1.3 Options

None.

C.1.4 Operands

file A pathname of a text file used for input. If no *file* operands are specified, the standard input shall be used.

C.1.5 External Influences

C.1.5.1 Standard Input

The standard input shall be used only if no *file* operands are specified. See Input Files.

C.1.5.2 Input Files

The input files shall be text files.

C.1.5.3 Environment Variables

The following environment variables shall affect the execution of `asa`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

C.1.5.4 Asynchronous Events

Default.

C.1.6 External Effects

C.1.6.1 Standard Output

The standard output shall be the text from the input file modified as described in C.1.2.

C.1.6.2 Standard Error

None.

C.1.6.3 Output Files

None.

C.1.7 Extended Description

None.

C.1.8 Exit Status

The `asa` utility shall exit with one of the following values:

0 All input files were output successfully.

>0 An error occurred.

C.1.9 Consequences of Errors

Default.

C.1.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The `asa` utility is needed to map ''standard'' FORTRAN 77 output into a form acceptable to contemporary printers. Usually `asa` is used to pipe data to the `lp` utility (see `lp` in 4.38.)

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.

History of Decisions Made

This utility is generally used only by FORTRAN programs. It was moved to this annex in response to multiple ballot objections requesting its removal. The working group decided to retain `asa` to avoid breaking the existing large base of FORTRAN applications that put carriage control characters in their output files. This is a compromise position to achieve balloting acceptance: the overhead of maintaining a separate option in POSIX. 2 for just this one utility is seen to be small in comparison to the benefit achieved for FORTRAN applications. Since it is a separate option, there is no requirement that a system have a FORTRAN compiler in order to run applications that need `asa`.

Historical implementations have used an ASCII `<form-feed>` character in response to a `'1'`, and an ASCII `<carriage-return>` in response to a `'+'`. It is suggested that implementations treat characters other than `'0'`, `'1'`, and `'+'` as `<space>` in the absence of any compelling reason to do otherwise. However, the action is listed here as ''unspecified,'' permitting an implementation to provide extensions to access fast multiple line slewing and channel seeking in a nonportable manner.

C.2 `fort77` — FORTRAN compiler

This utility is optional. It shall be provided on systems that support the FORTRAN Development Utilities Option.

C.2.1 Synopsis

`fort77` `[–c]` `[–g]` `[–L` *directory* `]` ... `[–O` *optlevel* `]` `[–o` *outfile* `]` `[–s]` `[–w]`
operand ...

C.2.2 Description

The `fort77` utility is the interface to the FORTRAN compilation system; it shall accept the full FORTRAN language defined by ISO 1539 {2}. 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`

shall 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 shall be written into the file named by the `–o` option (if present) or to the file `a.out`. The executable file shall be created as specified in 2.9.1.4, except that the file permissions shall be set to

S_IRWXO **|** S_IRWXG **|** S_IRWXU

(see POSIX.1 {8} 5.6.1.2) and that the bits specified by the *umask* of the process shall be cleared.

C.2.3 Options

The `fort77` utility shall conform to the utility argument syntax guidelines described in 2.10.2, 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.

— Conforming applications shall specify each option separately; that is, grouping option letters (e.g., `–cg`) need not be recognized by all implementations.

The following options shall be supported by the implementation:

−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-defined interactions with other options.

−s Produce object and/or executable files from which symbolic and other information not required for proper execution using the POSIX.1 {8} *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 shall be searched in the specified order. Implementations shall support at least ten instances of this option 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 optimization. If the *optlevel* option-argument is the digit 0, all special code optimizations shall be disabled. If it is the digit 1, the nature of the optimization is unspecified. If the −O option is omitted, the nature of the system's default optimization 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.

C.2.4 Operands

An *operand* is either in the form of a pathname or the form −l *library*. At least one operand of the pathname form shall be specified. The following operands shall be supported by the implementation:

file.f The pathname of a FORTRAN source file to be compiled and optionally passed to the link editor. The file name operand shall be of this form if the −c option is used.

file.a A library of object files typically produced by ar (see 6.1), and passed directly to the link editor. Implementations may recognize implementation-defined 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 recognize implementation-defined suffixes other than `.o` as denoting object files.

The processing of other files is implementation defined.

`−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 C.2.7. Implementations may recognize implementation-defined suffixes other than `.a` as denoting libraries.

C.2.5 External Influences

C.2.5.1 Standard Input

None.

C.2.5.2 Input Files

The input file shall 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 defined.

A `<tab>` character encountered within the first six characters on a line of source code shall cause the compiler to interpret the following character as if it were the seventh character on the line (i.e., in column 7).

C.2.5.3 Environment Variables

The following environment variables shall affect the execution of `fort77`:

LANG This variable shall determine the locale to use for the locale categories when both **LC_ALL** and the corresponding environment variable (beginning with **LC_**) do not specify a locale. See 2.6.

LC_ALL This variable shall determine the locale to be used to override any values for locale categories specified by the settings of **LANG** or any environment variables beginning with **LC_**.

LC_CTYPE This variable shall determine the locale for the interpretation of sequences of bytes of text data as characters (e.g., single- versus multibyte characters in arguments and input files).

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

TMPDIR This variable shall be interpreted as a pathname that should override the default directory for temporary files, if any.

C.2.5.4 Asynchronous Events

Default.

C.2.6 External Effects

C.2.6.1 Standard Output

None.

C.2.6.2 Standard Error

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 (nonzero) exit value.

C.2.6.3 Output Files

Object files, listing files, and/or executable files shall be produced in unspecified formats.

C.2.7 Extended Description

C.2.7.1 Standard Libraries

The `fort77` utility shall recognize the following `-l` operand for the standard library:

−l f This library contains all library functions referenced in ISO 1539 {2}. An implementation shall not require this operand 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 shall 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.

C.2.7.2 External Symbols

The FORTRAN compiler and link editor shall support the significance of external 1
symbols up to a length of at least 31 bytes. The compiler may fold case (i.e., may 1
ignore uppercase/lowercase distinctions between identifiers). The action taken 1
upon encountering symbols exceeding the implementation-defined maximum symbol length is unspecified.

The compiler and link editor shall 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-defined limit is exceeded; other actions are unspecified.

C.2.8 Exit Status

The fort77 utility shall exit with one of the following values:

0 Successful compilation or link edit.

>0 An error occurred.

C.2.9 Consequences of Errors

When fort77 encounters a compilation error, it shall write a diagnostic to standard error and continue to compile other source code operands. It shall return a nonzero exit status, but it is implementation defined whether an object module is created. If the link edit is unsuccessful, a diagnostic message shall be written to standard error, and fort77 shall exit with a nonzero status.

C.2.10 Rationale. *(This subclause is not a part of P1003.2)*

Examples, Usage

The following are examples of usage:

`fort77 -o foo xyz.f`	Compiles `xyz.f` and creates the executable `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 `a.out`.
`fort77 xyz.f b.o`	Compiles `xyz.f`, links it with `b.o`, and creates the executable `a.out`.

History of Decisions Made

The file inclusion and symbol definition (`#define`) mechanisms used by the `c89` utility were not included in POSIX.2—even though they are commonly implemented—since there is no requirement that the FORTRAN compiler use the C preprocessor.

The `−onetrip` option was not included in this specification, even though many historical compilers support it, because it is a relic from FORTRAN-66; it is an anachronism that should not be perpetuated.

Some implementations produce compilation listings. This aspect of FORTRAN has been left unspecified because there was opposition within the balloting group to the various methods proposed for implementing it: a `−V` option overlapped with historical vendor practice and a naming convention of creating files with `.l` suffixes collided with historical `lex` file naming practice.

There is no `−I` option in this version of POSIX.2 to specify a directory for file inclusion. An INCLUDE directive has been a part of the FORTRAN-8X discussions, but it is not clear whether it will be retained.

It is noted that many FORTRAN compilers produce an object module even when compilation errors occur; during a subsequent compilation, the compiler may patch the object module rather than recompiling all the code. Consequently, it is left to the implementor whether or not an object file is created.

The name of this utility was changed to `fort77` in Draft 9 to parallel the renaming of the C compiler. The name `f77` was not chosen to avoid collision with historical implementations.

A reference to MIL-STD-1753 was removed from an earlier draft in response to a request from the POSIX.9 working group. It was not the intention of this document to require certification of the FORTRAN compiler and the forthcoming POSIX.9 standard does not specify the military standard or any special preprocessing requirements. Furthermore, use of that document would have been inappropriate for an international standard.

The specification of optimization has been subject to changes through early drafts. At one time, `−O` and `−N` were Booleans: optimize and do not optimize (with an unspecified default). Some historical practice lead this to be changed to:

`-O 0` No optimization.

`-O 1` Some level of optimization.

`-O` *n* Other, unspecified levels of optimization.

It is not always clear whether ''good code generation'' is the same thing as optimization. Simple optimizations of local actions do not usually affect the semantics of a program. The `-O 0` option has been included to accommodate the very fussy nature of scientific calculations in a highly optimized environment; compilers make errors. Some degree of optimization is expected, even if it is not documented here, and the ability to shut it off completely could be important when porting an application. An implementation may treat `-O 0` as ''do less than normal'' if it wishes, but this is only meaningful if any of the operations it performs can affect the semantics of a program. It is highly dependent on the implementation whether doing less than normal makes sense. It is not the intent of this to ask for sloppy code generation, but rather to assure that any semantically visible optimization is suppressed.

The specification of standard library access is consistent with the C compiler specification. Implementations are not required to have `/usr/lib/libf.a`, as many historical implementations do, but if not they are required to recognize `'f'` as a token.

External symbol size limits are in a normative subclause; portable applications need to know these limits. However, the minimum maximum symbol length should be taken as a constraint on a portable application, not on an implementation, and consequently the action taken for a symbol exceeding the limit is unspecified. The minimum size for the external symbol table was added for similar reasons.

The Consequences of Errors subclause clearly specifies the compiler's behavior when compilation or link-edit error occur. The behavior of several historical implementations was examined, and the choice was made to be silent on the status of the executable, or `a.out`, file in the face of compiler or linker errors. If a linker writes the executable file, then links it on disk with *lseek* ()s and *write* ()s, the partially-linked executable can be left on disk and its execute bits turned off if the link edit fails. However, if the linker links the image in memory before writing the file to disk, it need not touch the executable file (if it already exists) because the link edit fails. Since both approaches are existing practice, a portable application shall rely on the exit status of `fort77`, rather than on the existence or mode of the executable file.

The `−g` and `−s` options are not specified as mutually exclusive. Historically these two options have been mutually exclusive, but because both are so loosely specified, it seemed cleaner to leave their interaction unspecified.

The requirement that portable applications specify compiler options separately is to reserve the multicharacter option namespace for vendor-specific compiler options, which are known to exist in many historical implementations. Implementations are not required to recognize, for example, `−gc` as if it were `−g −c`; nor are they forbidden from doing so. The synopsis shows all of the options separately to highlight this requirement on applications.

Echoing filenames to standard error is considered a diagnostic message, because it would otherwise difficult to associate an error message with the erring file. They are describing with ‘‘may’’ to allow implementations to use other methods of identifying files and to parallel the description in `c89`.

Annex D
(informative)

Bibliography

{B1} ISO 639: 1988, *Code for the representation of names of languages.* [1)]

{B2} ISO 2022: 1986, *Information processing— ISO 7-bit and 8-bit coded character sets—Code extension techniques.*

{B3} ISO 2047: 1975, *Information processing—Graphical representations for the control characters of the 7-bit coded character set.*

{B4} ISO 3166: 1988, *Code for the representation of names of countries.*

{B5} ISO 6429: 1988, *Information processing—Control functions for 7-bit and 8-bit coded character sets.*

{B6} ISO 6937-2: 1983, *Information processing—Coded character sets for text communication—Part 2: Latin alphabetic and non-alphabetic graphic characters.*

{B7} ISO 8802-3: 1989, *Information processing systems—Local area networks—Part 3: Carrier sense multiple access with collision detection (CSMA / CD) access method and physical layer specification.*

{B8} ISO 8806: 1988, *Data elements and interchange formats—Information interchange —Representation of dates and times.*

{B9} ISO 8859, *Information processing—8-bit single-byte coded graphic character sets.* (Parts 1 to 8 published.)

{B10} ISO/IEC 10367: . . . ,[2)] *Information processing—Repertoire of standardized coded graphic character sets for use in 8-bit codes.*

{B11} ISO/IEC 10646: . . . ,[3)] *Information technology—Universal Coded Character Set (UCS).*

---

1) ISO documents can be obtained from the ISO office, 1, rue de Varembé, Case Postale 56, CH-1211, Genève 20, Switzerland/Suisse.

2) To be approved and published.

3) To be approved and published.

{B12} International Organization for Standardization/Association Française de Normalisation. *Dictionary of Computer Science/Dictionnaire de L'Informatique.* Geneva/Paris: ISO/AFNOR, 1989.

{B13} ANSI X3.43-1986,[4] *Representations for Local Times of the Day for Information Interchange.*

{B14} GB 2312-1980, Chinese Association for Standardization. *Coded Chinese Graphic Character Set for Information Interchange.*

{B15} JIS X0208-1990, Japanese National Committee on ISO/IEC JTC 1/SC2. *Japanese Graphic Character Set for Information Interchange.*

{B16} JIS X0212-1990, Japanese National Committee on ISO/IEC JTC 1/SC2. *Supplementary Japanese Graphic Character Set for Information Interchange.*

{B17} KS C 5601-1987, Korean Bureau of Standards. *Korean Graphic Character Set for Information Interchange.*

{B18} IEEE Std 100-1988, *IEEE Standard Dictionary of Electrical and Electronics Terms.*

{B19} IEEE P1003.3,[5] *Standard for Information Technology—Test Methods for Measuring Conformance to POSIX*

{B20} IEEE P1003.3.2,[6] *Standard for Information Technology—Test Methods for Measuring Conformance to POSIX.2*

{B21} Aho, Alfred V., Kernighan, Brian W., Weinberger, Peter J., *The AWK Programming Language*, Reading, MA: Addison-Wesley, 1988.

{B22} Aho, Alfred V., Sethi, Ravi, Ullman, Jeffrey D., *Compilers, Principles, Techniques, and Tools*, Reading, MA: Addison-Wesley, 1986.

{B23} Aho, Alfred V., Ullman, Jeffrey D., *Principles of Compiler Design*, Reading, MA: Addison-Wesley, 1977.

{B24} American Telephone and Telegraph Company. *System V Interface Definition (SVID), Issues 2 and 3.* Morristown, NJ: UNIX Press, 1986, 1989.[7]

{B25} Bolsky, Morris I., Korn, David G., *The KornShell Command and Programming Language*, Englewood Cliffs, NJ: Prentice Hall, 1988.

---

4) ANSI documents can be obtained from the Sales Department, American National Standards Institute, 1430 Broadway, New York, NY 10018.

5) To be approved and published.

6) To be approved and published.

7) This is one of several documents that represent an industry specification in an area related to POSIX.2. The creators of such documents may be able to identify newer versions that may be interesting.

{B26} DeRemer, Frank, and Thomas J. Pennello, ''Efficient Computation of LALR(1) Look-ahead Sets.'' *SigPlan Notices* 15:8, 176-187, August, 1979.

{B27} Knuth, D. E. ''On the translation of languages from left to right.'' *Information and Control* 8:6, 607-639.

{B28} University of California at Berkeley—Computer Science Research Group. *4.3 Berkeley Software Distribution, Virtual VAX-11 Version.* Berkeley, CA: The Regents of the University of California, April 1986.

{B29} /usr/group Standards Committee. *1984 /usr/group Standard.* Santa Clara, CA: UniForum, 1984.

{B30} X/Open Company, Ltd. *X/Open Portability Guide, Issue 2.* Amsterdam: Elsevier Science Publishers, 1987.

{B31} X/Open Company, Ltd. *X/Open Portability Guide, Issue 3.* Englewood Cliffs, NJ: Prentice-Hall, 1989.

Annex E
(inform ative)
Rationale and Notes

This annex summarizes the deliberations of the IEEE P1003.2 Working Group, the committee charged by the IEEE Computer Society's Technical Committee on Operating Systems and Operational Environments with devising an interface standard for a shell and related utilities to support and extend POSIX. 1.

The annex is being published along with the standard to assist in the process of review. It contains historical information concerning the contents of the standard and why features were included or discarded by the Working Group. It also contains notes of interest to application programmers on recommended programming practices, emphasizing the consequences of some aspects of the standard that may not be immediately apparent.

Just as this standard relies on the knowledge of architecture, history, and definitions from the POSIX. 1, so does this annex. The reader is referred to the Rationale and Notes appendix of POSIX. 1 for background material and bibliographic information about UNIX systems in general and POSIX specifically, which will not be duplicated here.

E.1 General

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 1, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

E.1.1 Scope

E.1.2 Normative References

E.1.3 Conformance

E.2 Terminology and General Requirements

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 2, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

E.2.1 Conventions

E.2.2 Definitions

E.2.3 Built-in Utilities

E.2.4 Character Set

E.2.5 Locale

E.2.6 Environment Variables

E.2.7 Required Files

E.2.8 Regular Expression Notation

E.2.9 Dependencies on Other Standards

E.2.10 Utility Conventions

E.2.11 Utility Description Defaults

E.2.12 File Format Notation

E.2.13 Configuration Values

E.3 Shell Command Language

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 3, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

E.3.1 Shell Definitions

E.3.2 Quoting

E.3.3 Token Recognition

E.3.4 Reserved Words

E.3.5 Parameters and Variables

E.3.6 Word Expansions

E.3.7 Redirection

E.3.8 Exit Status for Commands

E.3.9 Shell Commands

E.3.10 Shell Grammar

E.3.11 Signals and Error Handling

E.3.12 Shell Execution Environment

E.3.13 Pattern Matching Notation

E.3.14 Special Built-in Utilities

E.4 Execution Environment Utilities

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 4, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting. Notations regarding utilities probably included in the UPE have been updated, without diff marks, based on the current working draft of 1003.2a.

Many utilities were evaluated by the working group; more utilities were excluded from the standard than included. The following list contains many common UNIX system utilities that were not included as Execution Environment Utilities or in one of the Software Development Environment groups. It is logistically difficult for this Rationale to correctly distribute the reasons for not including a utility among the various utility environment sections. Therefore, this section covers the reasons for all utilities not included in Sections 4 and 6 and Annexes A and C.

The working group started its deliberations with a recommended list of utilities provided by the X/Open group of companies. This list was a subset of the utilities in the *X/Open Portability Guide*, *Issue II*, so it was very closely related to System V. The list had already been purged of purely administrative utilities, such as those found in System V's Administered System Extension. Then, the working group applied its scope as a filter and substantially pruned the remaining list as well.

The following list of ''rejected'' utilities is limited by its historical roots; since the selected utilities emerged from primarily a System V base, this list does not include sometimes familiar entries from BSD. The working group received substantial input from representatives of the University of California at Berkeley and from companies that are firmly allied with BSD versions of the UNIX system, enough so that some BSD-derived utilities are included in the standard. However, this Rationale is now limited to a discussion of only those utilities actively or indirectly evaluated by the working group, rather than the list of all known UNIX utilities from all its variants. This list will most likely be augmented during the balloting process as balloters request specific rationales for their favorite commands.

In the list, the notation [*POSIX .2a*] is used to identify utilities that are being evaluated for inclusion in the forthcoming User Portability Extension to this standard. Similarly, [*POSIX .7*] is used for those that may be appropriate for the working group evaluating system administration and [*POSIX .Net*] for networking standards.

`adb` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. Furthermore, many useful aspects of `adb` are very hardware-specific.

`admin` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`as` Assemblers are hardware-specific and are included implicitly as part of the compilers in the standard.

`at` The `at` and `cron` family of utilities were omitted because portable applications could not rely on their behavior. [*POSIX .2a*]

`banner` The only known use of this command is as part of the LP printer header pages. It was decided that the format of the header is implementation defined, so this utility is superfluous to application portability.

`batch` The `at` and `cron` family of utilities were omitted because portable applications could not rely on their behavior. [*POSIX .2a*]

`cal` This calendar printing program is not useful to portable applications.

`calendar` This reminder service program is not useful to portable applications.

`cancel` The LP (line printer spooling) system specified is the most basic possible and did not need this level of application control. [*POSIX .7*]

`cflow` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.

`chroot` This is primarily of administrative use, requiring super-user privileges. [*POSIX .7*]

`col` No utilities defined in this standard produce output requiring such a filter. The `nroff` text formatter is present on many historical systems and will continue to remain as an extension; `col` is expected to be shipped by all the systems that ship `nroff`.

`cpio` This has been replaced by `pax`, for reasons explained in its own Rationale.

`cpp` Can be subsumed by `c89`.

`crontab` The `at` and `cron` family of utilities were omitted because portable applications could not rely on their behavior. [*POSIX .2a*]

`csplit` This utility's functionality can sometimes be provided by the `dd` or `sed` utilities (i.e., although these utilities cannot easily provide all of `csplit`'s features in one package, they can frequently be used for the type of task that `csplit` is being used for). [*POSIX .2a*]

`cu` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .Net*]

`cxref` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.

`dc` This utility's functionality can be provided by the `bc` utility; `bc` was selected because it was easier to use and had superior functionality. Although the historical versions of `bc` are implemented using `dc` as a base, this standard prescribes the interface and not the underlying mechanism used to implement it.

`delta` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`df` As the standard does not address the concept or nature of file systems, this command could not be specified in a manner useful to portable applications. [*POSIX .2a*]

`dircmp` Although a useful concept, the traditional output of this directory comparison program is not suitable for processing in applications programs. Also, the `diff -r` command gives equivalent functionality.

`dis` Disassemblers are hardware-specific.

`du` Because of differences between systems in measuring disk usage, this utility could not be used reliably by a portable application. [*POSIX .2a*]

`egrep` Marked obsolescent and replaced by the new version of `grep`.

`ex` This is typically a link to the `vi` terminal-oriented editor—not useful from shell scripts or typical application programs. The nonterminal oriented facilities of `ex` are provided by `ed`. [*POSIX .2a*]

`fgrep` Marked obsolescent and replaced by the new version of `grep`.

`file` Determining the type of file is generally accomplished with `test` or `find`. The added information available with `file` is of little use to a portable application, particularly since there is considerable variation in its output contents. [*POSIX .2a*]

`get` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`ld` Is subsumed by `c89`.

`line` The functionality of `line` can be provided with `read`.

`lint` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.

`login` Terminal oriented—not useful from shell scripts or typical application programs.

`lorder` This utility is an aid in creating an implementation-specific detail of object libraries that the working group did not feel required standardization.

`lpstat` The LP system specified is the most basic possible and did not need this level of application control. [*POSIX .7*]

`m4` The working group did not find that this macro processor had sufficiently wide usage for standardization.

`mail` This utility was omitted in favor of `mailx`, because there was a considerable functionality overlap between the two. The mail-sending aspects of `mailx` are covered in this standard, the mail-reading in the UPE. [*POSIX .2a*]

`mesg` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .2a*]

`mknod` This was omitted in favor of `mkfifo`, as `mknod` has too many implementation-defined functions. [*POSIX .7*]

`newgrp` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .2a*]

`news` Terminal oriented—not useful from shell scripts or typical application programs.

`nice` Due to historical variations in usage, and in the lack of underlying support from possible POSIX.1 {8} base systems, this cannot be used by applications to achieve reliable results. [*POSIX .2a*]

`nl` The useful functionality of `nl` can be provided with `pr`.

`nm` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. [*POSIX .2a*]

`pack` The working group found little interest in a portable data compression program (and there are others that are probably more widely used anyway).

`passwd` Terminal oriented—not useful from shell scripts or typical application programs. (There was also sentiment to avoid security-related utilities until requirements of 1003.6 are known.)

`pcat` The working group found little interest in a portable data compression program (and there are others that are probably more widely used anyway).

`pg` Terminal oriented—not useful from shell scripts or typical application programs.

`prof` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.

`prs` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`ps` This utility has historically been difficult to specify portably due to the many implementation-defined aspects of processes. Furthermore, a portable application can rarely rely on information about what other processes are doing, as security mechanisms may prevent it. A process requiring one of its children's process IDs (such as for use with the `kill` command) will have to record the IDs at the time of creation. [*POSIX .2a*]

`red` Restricted editor. This was not considered by the working group because it never provided the level of security restriction required.

`rmdel` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`rsh` Restricted shell. This was not considered by the working group
because it does not provide the level of security restriction that is 1
implied by historical documentation. 1

`sact` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`sdb` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool. Furthermore, some useful aspects of `sdb` are very hardware-specific.

`sdiff` The ''side-by-side `diff`'' utility from System V was omitted because it is used infrequently, and even less so by portable applications. Despite being in System V, it is not in the *SVID* or *XPG* .

`shar` Utilities with this type of functionality (‘‘shell-based archivers’’) are in wide use, despite not being included in System V or BSD systems. However, the working group felt this sort of program was more widely used by human users than portable applications.

`shl` Terminal oriented—not useful from shell scripts or typical application programs. The job control aspects of the Shell Command Language are generally more useful and are being evaluated for the UPE.

`size` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This utility is primarily a debugging tool.

`spell` Not useful from shell scripts or typical application programs.

`split` The functionality can sometimes be provided by the `dd`, `sed`, or (for some uses) `xargs` utilities (i.e., although these utilities cannot easily provide all of `split`'s features in one package, they can sometimes be used for the type of task that `split` is being used for). [*POSIX .2a*]

`strings` This is normally used by human users during debugging, rather than by applications. [*POSIX .2a*]

`su` Not useful from shell scripts or typical application programs. (There was also sentiment to avoid security-related utilities until requirements of POSIX. 6 are known.)

`sum` This utility was renamed `cksum`.

`tabs` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .2a*]

`time` Not necessary for portable applications. It is frequently used by human users in debugging or for informal benchmarks. It is doubtful whether any standardized definitions of the output could be agreed upon.

`tsort` This utility is an aid in creating an implementation-specific detail of object libraries that the working group did not feel required standardization.

`unget` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`unpack` The working group found little interest in a portable data compression program (and there are others that are probably more widely used anyway).

`uucp`
`uulog`
`uupick`
`uustat`
`uuto` The UUCP utilities and their protocol description were removed 1
from an early draft because responsibility for them was officially 1
requested by the POSIX group developing networking interfaces. 1

`val` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`vi` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .2a*]

`wall` Terminal oriented—not useful from shell scripts or typical application programs. It is generally used by system administrators, as well. [*POSIX .7*]

`what` The intent of the various software development utilities was to assist in the installation (rather than the actual development and debugging) of applications. This SCCS utility is primarily a development tool.

`who` The ability to determine other users on the system was felt to be at risk in a trusted implementation, so its use could not be considered by a portable application. [*POSIX .2a*]

`write` Terminal oriented—not useful from shell scripts or typical application programs. [*POSIX .2a*]

E.4.1 awk — Pattern scanning and processing language

E.4.2 basename — Return nondirectory portion of pathname

E.4.3 bc — Arbitrary-precision arithmetic language

E.4.4 cat — Concatenate and print files

E.4.5 cd — Change working directory

E.4.6 chgrp — Change file group ownership

E.4.7 chmod — Change file modes

E.4.8 chown — Change file ownership

E.4.9 cksum — Write file checksums and block counts

E.4.10 cmp — Compare two files

E.4.11 comm — Select or reject lines common to two files

E.4.12 command — Select or reject lines common to two files

E.4.13 cp — Copy files

E.4.14 cut — Cut out selected fields of each line of a file

E.4.15 date — Write the date and time

E.4.16 dd — Convert and copy a file

E.4.17 diff — Compare two files

E.4.18 dirname — Return directory portion of pathname

E.4.19 echo — Write arguments to standard output

E.4.20 ed — Edit text

E.4.21 env — Set environment for command invocation

E.4.22 `expr` — Evaluate arguments as an expression

E.4.23 `false` — Return false value

E.4.24 `find` — Find files

E.4.25 `fold` — Filter for folding lines

E.4.26 `getconf` — Get configuration values

E.4.27 `getopts` — Parse utility options

E.4.28 `grep` — File pattern searcher

E.4.29 `head` — Copy the first part of files

E.4.30 `id` — Return user identity

E.4.31 `join` — Relational database operator

E.4.32 `kill` — Terminate or signal processes

E.4.33 `ln` — Link files

E.4.34 `locale` — Get locale-specific information

E.4.35 `localedef` — Define locale environment

E.4.36 `logger` — Log messages

E.4.37 `logname` — Return user's login name

E.4.38 `lp` — Send files to a printer

E.4.39 `ls` — List directory contents

E.4.40 `mailx` — Process messages

E.4.41 `mkdir` — Make directories

E.4.42 `mkfifo` — Make FIFO special files

E.4.43 `mv` — Move files

E.4.44 `nohup` — Invoke a utility immune to hangups

E.4.45 `od` — Dump files in various formats

E.4.46 `paste` — Merge corresponding or subsequent lines of files

E.4.47 `pathchk` — Check pathnames

E.4.48 `pax` — Portable archive interchange

E.4.49 `pr` — Print files

E.4.50 `printf` — Write formatted output

E.4.51 `pwd` — Return working directory name

E.4.52 `read` — Read a line from standard input

E.4.53 `rm` — Remove directory entries

E.4.54 `rmdir` — Remove directories

E.4.55 `sed` — Stream editor

E.4.56 `sh` — Shell, the standard command language interpreter

E.4.57 `sleep` — Suspend execution for an interval

E.4.58 `sort` — Sort, merge, or sequence check text files

E.4.59 `stty` — Set the options for a terminal

E.4.60 `tail` — Copy the last part of a file

E.4.61 `tee` — Duplicate standard input

E.4.62 `test` — Evaluate expression

E.4.63 `touch` — Change file access and modification times

E.4.64 `tr` — Translate characters

E.4.65 `true` — Return true value

E.4.66 `tty` — Return user's terminal name

E.4.67 `umask` — Get or set the file mode creation mask

E.4.68 `uname` — Return system name

E.4.69 `uniq` — Report or filter out repeated lines in a file

E.4.70 `wait` — Await process completion

E.4.71 `wc` — Word, line, and byte count

E.4.72 `xargs` — Construct argument list(s) and invoke utility

E.5 User Portability Utilities Option

Editor's Note: This section is unused in this revision of the standard.

E.6 Software Development Utilities Option

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 6, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

This is the first of the optional utility environments. The working group decided there were two basic classes of systems to be supported: general application execution and software development. The first is widely used and is the primary reason for the development of this standard. The second, however, represents only a (small?) subset of the first; the users are generally only those who are developing or installing C or FORTRAN applications.

Therefore, all the development environments are optional, giving users the option of specifying a smaller, (presumably) less expensive system. There are three separate optional environments, so that C-only or FORTRAN -only users do not have to specify unneeded components. As further languages are supported by this standard, their environments will also be optional.

An implementation must provide all three of these utilities to claim conformance to this section.

See section **E.4** for a discussion of utilities excluded from this group.

E.6.1 `ar` — Create and maintain library archives

E.6.2 `make` — Maintain, update, and regenerate groups of programs

E.6.3 `strip` — Remove unnecessary information from executable files

E.7 Language-Independent System Services

Editor's Note: The text of the Rationale for this section has been temporarily located in Section 7, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

E.7.1 Shell Command Interface

E.7.2 Access Environment Variables

E.7.3 Regular Expression Matching

E.7.4 Pattern Matching

E.7.5 Command Option Parsing

E.7.6 Generate Pathnames Matching a Pattern

E.7.7 Perform Word Expansions

E.7.8 Get POSIX Configurable Variables

E.7.9 Locale Control

E.8 C Language Development Utilities Option

Editor's Note: The text of the Rationale for this section has been temporarily located in Annex A, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

This is the second of the optional utility environments.

An implementation must provide all three of these utilities to claim conformance to this section.

See section **E.4** for a discussion of utilities excluded from this group.

E.8.1 `c89` — Compile Standard C programs

E.8.2 `lex` — Generate programs for lexical tasks

E.8.3 `yacc` — Yet another compiler compiler

E.9 C Language Bindings Option

Editor's Note: The text of the Rationale for this section has been temporarily located in Annex B, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

E.9.1 C Language Definitions

E.9.2 C Numerical Limits

E.9.3 C Binding for Shell Command Interface

E.9.4 C Binding for Access Environment Variables

E.9.5 C Binding for Regular Expression Matching

E.9.6 C Binding for Match Filename or Pathname

E.9.7 C Binding for Command Option Parsing

E.9.8 C Binding for Generate Pathnames Matching a Pattern

E.9.9 C Binding for Perform Word Expansions

E.9.10 C Binding for Get POSIX Configurable Variables

E.9.11 C Binding for Locale Control

E.10 FORTRAN Development and Runtime Utilities Options

Editor's Note: The text of the Rationale for this section has been temporarily located in Annex C, adjacent to the text it is explaining. The text will return to this annex after the completion of balloting.

This is the third and fourth of the optional utility environments.

See section **E.4** for a discussion of utilities excluded from this group.

E.10.1 `asa` — Interpret carriage control characters

E.10.2 `fort77` — FORTRAN compiler

Annex F
(informative)
Sample National Profile

Editor's Note: All uses of the term ''character set'' this annex have been changed to ''coded character set'' without further diff marks.

This annex is an example of a country's needs with respect to this standard and how those needs relate to other international standards as well as national standards. The example provided is included here for informative purposes and is not a formal standard in the country in question. It is provided by the Danish Standards Association [1)] and is as accurate as possible with regards to Danish needs.

F.1 (Example) Danish National Profile

This is the definition of the Danish Standards Association POSIX.2 profile. The subset of conforming implementations that provide the required characteristics below is referred to as conforming to the ''Danish Standards Association (DS) Environment Profile'' for this standard.

This profile specifies the following requirements on implementations:

(1) In POSIX.2 section 2.13.1, the limit {COLL_WEIGHTS_MAX} shall be provided with a value of 4. All other limits shall conform to at least the minimum values shown in Table 2-16.

(2) The following options shall be supported according to POSIX.2 section 2.13.2:

---

1) Further information may be obtained from the Danish Standards Association, Attn: S142u22A8 Baunegaardsvej 73, DK-2900 Hellerup, Denmark; FAX: +45 39 77 02 02; Email: `u22a8@dkuug.dk`

The data is also available electronically by anonymous FTP or FTAM at the site `dkuug.dk` in the directory `i18n`, where some other example national profiles, locales, and *charmaps* may also be found. They are also available by an archive server reached at `archive@dkuug.dk`; use '`Subject: help`'' for further information.

`POSIX2_C_BIND`	Optional.	2
`POSIX2_C_DEV`	Optional.	2
`POSIX2_FORT_DEV`	Optional.	2
`POSIX2_FORT_RUN`	Optional.	2
`POSIX2_LOCALEDEF`	Required; the system shall support the creation of locales as described in 4.35.	2
`POSIX2_SW_DEV`	Optional.	2

F.1.1 Danish Locale Model

Editor's Note: This subclause is offered as rationale for the current state of this example annex. It will not necessarily appear in this form in any final version of the annex.

Creating a national locale for Denmark has been a quite elaborate effort. Time and again, we thought we had reached an agreement on the locale, but then some aspect disrupted the entire work, and we more or less had to start all over.

We think we have identified the cause of these problems to a general uncertainty regarding the exact purpose of a ''national'' locale. If we look at the Danish situation (which we know pretty well by now), we have identified several levels of locales, depending on the ''complexity'' of the collating sequence (or more generally sorting different kinds of text):

(1) *Byte/machine level.* Here everything is sorted according to the character's byte value.

(2) *Character/utility level.* Here we want to work almost on the same level as (1), i.e., character by character, but obeying a (simple) collating sequence that ensures that, for example, upper- and lowercase letters are equivalent, or that national characters are sorted correctly. The characters still do not have any ''implicit'' meaning, and the comparison of two strings is still deterministic; i.e., strings that are different at level 1 are still different at level 2.

(3) *Text/application level.* Here we want to be able to search in text looking for specific words or items. The comparison is still performed on a character-by-character basis, but possibly ignoring some characters that are not important, and determinism is not important either.

(4) *Semantic/dictionary/library/phone-book level.* Entire words like ''the'' are omitted from comparisons; maybe soundex is required. This probably requires specially developed software.

Our problem has been the conflicting requirements from each of these levels, which we optimistically have tried to combine into a single national locale (ignoring level 4, however). The POSIX Locale is aimed at level 2; i.e., at a rather low level. Many of our attempts to write a national Danish locale have failed because we have actually tried to write a level 3 locale, and finding that it did not work as an alternative to the default POSIX locale at level 2.

The locale we now provide is the final compromise between level 2 and level 3, by taking our latest attempt aimed at level 3, and make the comparison completely deterministic, and thus bring it down to level 2.

We also have found that we may need to include some more information in the identification of a specific locale than just the country code, the language code, and the coded character set, since what we have had most problems with was the purpose or scope of a specific locale; i.e., is it just a nationalized version of the POSIX Locale (e.g., extended with `<ae>`, `<o/>`, and `<aa>` at the proper positions), is it aimed at text search (ignoring certain characters), or is it on an even higher level? Many such alternative locales would certainly be useful for various classes of problems or applications, so our model for the locale name identification string includes a *<version>* parameter.

We hope by providing these comments to have clarified our intention with the locale definitions to save other countries from doing our mistakes all over.

F.2 Locale String Definition Guideline

The following guideline is used for specifying the locale identification string: [2)]

> `"%2.2s_%2.2s.%s,%s"`, *<language>*, *<territory>*, *<coded-character-set>*, *<version>*

where *<language>* shall be taken from ISO 639 {B1} and *<territory>* shall be the two-letter country code of ISO 3166 {B4}, if possible. The *<language>* shall be specified with lowercase letters only, and the *<territory>* shall be specified in uppercase letters only. An optional *<coded-character-set>* specification may follow after a `<period>` for the name of the coded character set; if just a numeric specification is present, this shall represent the number of the international standard describing the coded character set. If the *<coded-character-set>* specification is not present, the encoded character-set-specific locale shall be determined by the **CHARSET** environment variable, and if this is unset or null, the encoding of ISO 8859-1 {5} shall be assumed. A parameter specifying a *<version>* of the locale may be placed after the optional *<coded-character-set>* specification, delimited by `<comma>`. This may be used to discriminate between different cultural needs; for instance, dictionary order versus a more systems-oriented collating order.

---

2) The guideline was inspired by the *X/Open Portability Guide* {B31}.

F.3 Scope of Danish National Locale

This national locale covers the Danish language in Denmark. In addition, Faroese and Greenlandic `LC_TIME` and `LC_MESSAGES` specifications have been defined; the rest of the Danish national locale shall be used for these locales as well.

This locale is designed to be coded character-set independent. It completely specifies the behavior of systems based on ISO/IEC 10646 {B11} (with ISO 6429 {B5} control character encoding) together with many 7-bit and 8-bit encoded character sets, including ISO 8859 character sets and major vendor-specific 8-bit character sets (with ISO 6429 {B5} or ISO/IEC 646 {1} control character encoding when applicable).

This locale is portable as long as the character naming in the charmap description file `ISO_10646` for ISO/IEC 10646 {B11} is followed. Examples of such charmap files for ISO/IEC 10646 {B11} and ISO 8859-1 {5} are shown in F.5.1 and F.5.2.

The collating sequence is completely deterministic and is aimed for usage in system tools. Other Danish collation sequences with nondeterministic properties, which may be needed for some application programs, are not covered by this locale.

The `LC_TYPE` category of the locale is quite general and may be useful for other locales; also the `LC_COLLATE` category, though specifically Danish, may be a good template from which to generate other locales.

Following the preceding guidelines for locale names, the national Danish locale string shall be:

```
da_DK
```

F.3.1 `da_DK` — (Example) Danish National Locale

```
escape_char     /
comment_char    %                                                              1
% Danish example national locale for the language Danish                       1
% Source: Danish Standards Association                                         1
% Revision 1.7 1991-05-07                                                      1

LC_CTYPE                                                                       1

digit   <0>;<1>;<2>;<3>;<4>;<5>;<6>;<7>;<8>;<9>                                1

xdigit  <0>;<1>;<2>;<3>;<4>;<5>;<6>;<7>;<8>;<9>;/                              1
        <A>;<B>;<C>;<D>;<E>;<F>;<a>;<b>;<c>;<d>;<e>;<f>                        1

blank   <SP>;<HT>;<NS>                                                         1

space   <SP>;<LF>;<VT>;<FF>;<CR>;<HT>;<NS>                                     1

upper   <A>;<B>;<C>;<D>;<E>;<F>;<G>;<H>;<I>;<J>;/                              1
        <K>;<L>;<M>;<N>;<O>;<P>;<Q>;<R>;<S>;<T>;/                              1
        <U>;<V>;<W>;<X>;<Y>;<Z>;<A!>;<A’>;<A/>>;<A?>;/                         1
```

```
          <A:>;<AA>;<AE>;<C,>;<E!>;<E'>;<E/>>;<E:>;<I!>;<I'>;/          1
          <I/>>;<I:>;<D->;<N?>;<O!>;<O'>;<O/>>;<O?>;<O:>;<O//>;/        1
          <U!>;<U'>;<U/>>;<U:>;<Y'>;<TH>;<A->;<C/>>;<C.>;<E->;/         1
          <E.>;<G/>>;<G(>;<G.>;<G,>;<H/>>;<I?>;<I->;<I.>;<I;>;/         1
          <J/>>;<K,>;<H//>;<IJ>;<L.>;<L,>;<N,>;<OE>;<O->;<T//>;/        1
          <NG>;<A;>;<L//>;<L<>;<S'>;<S/>>;<S<>;<S,>;<T<>;<Z'>;/         1
          <Z<>;<Z.>;<R'>;<R,>;<A(>;<L'>;<C'>;<C<>;<E;>;<E<>;/           1
          <D<>;<D//>;<N'>;<N<>;<U?>;<O">;<U->;<U(>;<R<>;<U0>;/          1
          <U;>;<U">;<W/>>;<Y/>>;<T,>;<Y:>;<A<>;<A_>;<'A>;<A1>;/
          <A2>;<A3>;<B.>;<B_>;<D_>;<D.>;<D;>;<E(>;<E_>;<E?>;/
          <F.>;<G<>;<G->;<G//>;<H:>;<H.>;<H,>;<H;>;<I<>;<I(>;/
          <J(>;<K'>;<K<>;<K_>;<K.>;<K;>;<L_>;<M'>;<M.>;<N.>;/
          <N_>;<O<>;<O(>;<O_>;<O;>;<O1>;<P'>;<R.>;<R_>;<S.>;/
          <S;>;<T_>;<T.>;<U<>;<V?>;<W'>;<W.>;<W:>;<X.>;<X:>;/
          <Y!>;<Y.>;<Z/>>;<Z(>;<Z_>;<Z//>;<EZ>;<G'>;<'B>;<'D>;/
          <'G>;<'J>;<'Y>;<ED>;<IO>;<D%>;<G%>;<IE>;<DS>;<II>;/
          <YI>;<J%>;<LJ>;<NJ>;<Ts>;<KJ>;<V%>;<DZ>;<A=>;<B=>;/
          <V=>;<G=>;<D=>;<E=>;<Z%>;<Z=>;<I=>;<J=>;<K=>;<L=>;/
          <M=>;<N=>;<O=>;<P=>;<R=>;<S=>;<T=>;<U=>;<F=>;<H=>;/
          <C=>;<C%>;<S%>;<Sc>;<=">;<Y=>;<%">;<JE>;<JU>;<JA>;/
          <I3>;<A%>;<E%>;<Y%>;<I%>;<O%>;<U%>;<W%>;<A*>;<B*>;/
          <G*>;<D*>;<E*>;<Z*>;<Y*>;<H*>;<I*>;<K*>;<L*>;<M*>;/
          <N*>;<C*>;<O*>;<P*>;<R*>;<S*>;<T*>;<U*>;<F*>;<X*>;/
          <Q*>;<W*>;<J*>;<V*>

lower     <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>;<ss>;<a!>;<a'>;<a/>>;/
          <a?>;<a:>;<aa>;<ae>;<c,>;<e!>;<e'>;<e/>>;<e:>;<i!>;/
          <i'>;<i/>>;<i:>;<d->;<n?>;<o!>;<o'>;<o/>>;<o?>;<o:>;/
          <o//>;<u!>;<u'>;<u/>>;<u:>;<y'>;<th>;<y:>;<a->;<c/>>;/
          <c.>;<e->;<e.>;<g/>>;<g(>;<g.>;<g,>;<h/>>;<i?>;<i->;/
          <'n>;<kk>;<i;>;<j/>>;<k,>;<h//>;<i.>;<ij>;<l.>;<l,>;/
          <n,>;<oe>;<o->;<t//>;<ng>;<a;>;<l//>;<l<>;<s'>;<s/>>;/
          <s<>;<s,>;<t<>;<z'>;<z<>;<z.>;<r'>;<r,>;<a(>;<l'>;/
          <c'>;<c<>;<e;>;<e<>;<d<>;<d//>;<n'>;<n<>;<u?>;<o">;/
          <u->;<u(>;<r<>;<u0>;<u;>;<u">;<w/>>;<y/>>;<t,>;<a<>;/         1
          <a_>;<'a>;<a1>;<a2>;<a3>;<b.>;<b_>;<d_>;<d.>;<d;>;/
          <e(>;<e_>;<e?>;<f.>;<g<>;<g->;<g//>;<h:>;<h.>;<h,>;/
          <h;>;<i<>;<i(>;<j(>;<k'>;<k<>;<k_>;<k.>;<k;>;<l_>;/
          <m'>;<m.>;<n.>;<n_>;<o<>;<o(>;<o_>;<o;>;<o1>;<p'>;/
          <r.>;<r_>;<s.>;<s;>;<t_>;<t.>;<u<>;<v?>;<w'>;<w.>;/
          <w:>;<x.>;<x:>;<y!>;<y.>;<z/>>;<z(>;<z_>;<z//>;<ez>;/
          <g'>;<'b>;<'d>;<'g>;<'j>;<'y>;<ed>;<nS>;<sB>;<a=>;/
          <b=>;<v=>;<g=>;<d=>;<e=>;<z%>;<z=>;<i=>;<j=>;<k=>;/
          <l=>;<m=>;<n=>;<o=>;<p=>;<r=>;<s=>;<t=>;<u=>;<f=>;/
          <h=>;<c=>;<c%>;<s%>;<sc>;<='>;<y=>;<%'>;<je>;<ju>;/
          <ja>;<io>;<d%>;<g%>;<ie>;<ds>;<ii>;<yi>;<j%>;<lj>;/
          <nj>;<ts>;<kj>;<v%>;<dz>;<a%>;<e%>;<y%>;<i%>;<a*>;/
          <b*>;<g*>;<d*>;<e*>;<z*>;<y*>;<h*>;<i*>;<k*>;<l*>;/
          <m*>;<n*>;<c*>;<o*>;<p*>;<r*>;<*s>;<s*>;<t*>;<u*>;/
          <f*>;<x*>;<q*>;<w*>;<j*>;<v*>;<o%>;<u%>;<w%>;<A5>;/
          <I5>;<U5>;<E5>;<O5>;<tU>;<yA>;<yU>;<yO>;<wA>;<a6>;/
          <i6>;<u6>;<e6>;<o6>;<TU>;<YA>;<YU>;<YO>;<WA>;<KA>;/
          <KE>;<ff>;<fi>;<fl>;<ft>;<st>

alpha     <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>;<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>;<-->;<A!>;<A’>;<A/>>;<A?>;<A:>;<AA>;<AE>;/
        <C,>;<E!>;<E’>;<E/>>;<E:>;<I!>;<I’>;<I/>>;<I:>;<D->;/
        <N?>;<O!>;<O’>;<O/>>;<O?>;<O:>;<O//>;<U!>;<U’>;<U/>>;/
        <U:>;<Y’>;<TH>;<ss>;<a!>;<a’>;<a/>>;<a?>;<a:>;<aa>;/
        <ae>;<c,>;<e!>;<e’>;<e/>>;<e:>;<i!>;<i’>;<i/>>;<i:>;/
        <d->;<n?>;<o!>;<o’>;<o/>>;<o?>;<o:>;<o//>;<u!>;<u’>;/
        <u/>>;<u:>;<y’>;<th>;<y:>;<A->;<C/>>;<C.>;<E->;<E.>;/
        <G/>>;<G(>;<a->;<c/>>;<c.>;<e->;<e.>;<g/>>;<g(>;<G.>;/
        <G,>;<H/>>;<I?>;<I->;<I.>;<g.>;<g,>;<h/>>;<i?>;<i->;/
        <I;>;<J/>>;<K,>;<H//>;<IJ>;<L.>;<L,>;<N,>;<OE>;<O->;/
        <T//>;<NG>;<’n>;<kk>;<i;>;<j/>>;<k,>;<h//>;<i.>;<ij>;/
        <l.>;<l,>;<n,>;<oe>;<o->;<t//>;<ng>;<A;>;<L//>;<L<>;/
        <S’>;<S/>>;<S<>;<S,>;<T<>;<Z’>;<Z<>;<Z.>;<a;>;<l//>;/
        <l<>;<s’>;<s/>>;<s<>;<s,>;<t<>;<z’>;<z<>;<z.>;<R’>;/
        <R,>;<A(>;<L’>;<C’>;<C<>;<E;>;<E<>;<D<>;<D//>;<N’>;/
        <N<>;<U?>;<O">;<U->;<U(>;<R<>;<U0>;<U;>;<U">;<W/>>;/        1
        <Y/>>;<T,>;<Y:>;<r’>;<r,>;<a(>;<l’>;<c’>;<c<>;<e;>;/
        <e<>;<d<>;<d//>;<n’>;<n<>;<u?>;<o">;<u->;<u(>;<r<>;/
        <u0>;<u;>;<u">;<w/>>;<y/>>;<t,>;<a<>;<A<>;<a_>;<A_>;/       1
        <’a>;<’A>;<a1>;<A1>;<a2>;<A2>;<a3>;<A3>;<b.>;<B.>;/
        <b_>;<B_>;<d_>;<D_>;<d.>;<D.>;<d;>;<D;>;<e(>;<E(>;/
        <e_>;<E_>;<e?>;<E?>;<f.>;<F.>;<g<>;<G<>;<g->;<G->;/
        <g//>;<G//>;<h:>;<H:>;<h.>;<H.>;<h,>;<H,>;<h;>;<H;>;/
        <i<>;<I<>;<i(>;<I(>;<j(>;<J(>;<k’>;<K’>;<k<>;<K<>;/
        <k_>;<K_>;<k.>;<K.>;<k;>;<K;>;<l_>;<L_>;<m’>;<M’>;/
        <m.>;<M.>;<n.>;<N.>;<n_>;<N_>;<o<>;<O<>;<o(>;<O(>;/
        <o_>;<O_>;<o;>;<O;>;<o1>;<O1>;<p’>;<P’>;<r.>;<R.>;/
        <r_>;<R_>;<s.>;<S.>;<s;>;<S;>;<t_>;<T_>;<t.>;<T.>;/
        <u<>;<U<>;<v?>;<V?>;<w’>;<W’>;<w.>;<W.>;<w:>;<W:>;/
        <x.>;<X.>;<x:>;<X:>;<y!>;<Y!>;<y.>;<Y.>;<z/>>;<Z/>>;/
        <z(>;<Z(>;<z_>;<Z_>;<z//>;<Z//>;<ez>;<EZ>;<g’>;<G’>;/
        <’b>;<’B>;<’d>;<’D>;<’g>;<’G>;<’j>;<’J>;<’y>;<’Y>;/
        <ed>;<ED>;<nS>;<IO>;<D%>;<G%>;<IE>;<DS>;<II>;<YI>;/
        <J%>;<LJ>;<NJ>;<Ts>;<KJ>;<V%>;<DZ>;<A=>;<B=>;<V=>;/
        <G=>;<D=>;<E=>;<Z%>;<Z=>;<I=>;<J=>;<K=>;<L=>;<M=>;/
        <N=>;<O=>;<P=>;<R=>;<S=>;<T=>;<U=>;<F=>;<H=>;<C=>;/
        <C%>;<S%>;<Sc>;<=">;<Y=>;<%">;<JE>;<JU>;<JA>;<a=>;/
        <b=>;<v=>;<g=>;<d=>;<e=>;<z%>;<z=>;<i=>;<j=>;<k=>;/
        <l=>;<m=>;<n=>;<o=>;<p=>;<r=>;<s=>;<t=>;<u=>;<f=>;/
        <h=>;<c=>;<c%>;<s%>;<sc>;<=’>;<y=>;<%’>;<je>;<ju>;/
        <ja>;<io>;<d%>;<g%>;<ie>;<ds>;<ii>;<yi>;<j%>;<lj>;/
        <nj>;<ts>;<kj>;<v%>;<dz>;<I3>;<A%>;<E%>;<Y%>;<I%>;/
        <O%>;<U%>;<W%>;<A*>;<B*>;<G*>;<D*>;<E*>;<Z*>;<Y*>;/
        <H*>;<I*>;<K*>;<L*>;<M*>;<N*>;<C*>;<O*>;<P*>;<R*>;/
        <S*>;<T*>;<U*>;<F*>;<X*>;<Q*>;<W*>;<J*>;<V*>;<a%>;/
        <e%>;<y%>;<i%>;<a*>;<b*>;<g*>;<d*>;<e*>;<z*>;<y*>;/
        <h*>;<i*>;<k*>;<l*>;<m*>;<n*>;<c*>;<o*>;<p*>;<r*>;/
        <*s>;<s*>;<t*>;<u*>;<f*>;<x*>;<q*>;<w*>;<j*>;<v*>;/
        <o%>;<u%>;<w%>;<p+>;<v+>;<gf>;<H’>;<aM>;<aH>;<wH>;/
        <ah>;<yH>;<a+>;<b+>;<tm>;<t+>;<tk>;<g+>;<hk>;<x+>;/
        <d+>;<dk>;<r+>;<z+>;<s+>;<sn>;<c+>;<dd>;<tj>;<zH>;/
        <e+>;<i+>;<f+>;<q+>;<k+>;<l+>;<m+>;<n+>;<h+>;<w+>;/
        <j+>;<y+>;<A+>;<B+>;<G+>;<D+>;<H+>;<W+>;<Z+>;<X+>;/
```

```
          <Tj>;<J+>;<K%>;<K+>;<L+>;<M%>;<M+>;<N%>;<N+>;<S+>;/
          <E+>;<P%>;<P+>;<Zj>;<ZJ>;<Q+>;<R+>;<Sh>;<T+>;<b4>;/
          <p4>;<m4>;<f4>;<d4>;<t4>;<n4>;<l4>;<g4>;<k4>;<h4>;/
          <j4>;<q4>;<x4>;<zh>;<ch>;<sh>;<r4>;<z4>;<c4>;<s4>;/
          <a4>;<o4>;<e4>;<eh>;<ai>;<ei>;<au>;<ou>;<an>;<en>;/
          <aN>;<eN>;<er>;<i4>;<u4>;<iu>;<A5>;<a5>;<I5>;<i5>;/
          <U5>;<u5>;<E5>;<e5>;<O5>;<o5>;<ka>;<ga>;<ki>;<gi>;/
          <ku>;<gu>;<ke>;<ge>;<ko>;<go>;<sa>;<za>;<si>;<zi>;/
          <su>;<zu>;<se>;<ze>;<so>;<zo>;<ta>;<da>;<ti>;<di>;/
          <tU>;<tu>;<du>;<te>;<de>;<to>;<do>;<na>;<ni>;<nu>;/
          <ne>;<no>;<ha>;<ba>;<pa>;<hi>;<bi>;<pi>;<hu>;<bu>;/
          <pu>;<he>;<be>;<pe>;<ho>;<bo>;<po>;<ma>;<mi>;<mu>;/
          <me>;<mo>;<yA>;<ya>;<yU>;<yu>;<yO>;<yo>;<ra>;<ri>;/
          <ru>;<re>;<ro>;<wA>;<wa>;<wi>;<we>;<wo>;<n5>;<a6>;/
          <A6>;<i6>;<I6>;<u6>;<U6>;<e6>;<E6>;<o6>;<O6>;<Ka>;/
          <Ga>;<Ki>;<Gi>;<Ku>;<Gu>;<Ke>;<Ge>;<Ko>;<Go>;<Sa>;/
          <Za>;<Si>;<Zi>;<Su>;<Zu>;<Se>;<Ze>;<So>;<Zo>;<Ta>;/
          <Da>;<Ti>;<Di>;<TU>;<Tu>;<Du>;<Te>;<De>;<To>;<Do>;/
          <Na>;<Ni>;<Nu>;<Ne>;<No>;<Ha>;<Ba>;<Pa>;<Hi>;<Bi>;/          1
          <Pi>;<Hu>;<Bu>;<Pu>;<He>;<Be>;<Pe>;<Ho>;<Bo>;<Po>;/          1
          <Ma>;<Mi>;<Mu>;<Me>;<Mo>;<YA>;<Ya>;<YU>;<Yu>;<YO>;/
          <Yo>;<Ra>;<Ri>;<Ru>;<Re>;<Ro>;<WA>;<Wa>;<Wi>;<We>;/
          <Wo>;<N6>;<Vu>;<KA>;<KE>;<ff>;<fi>;<fl>;<ft>;<st>;/
          <yf>

cntrl     <NU>;<SH>;<SX>;<EX>;<ET>;<EQ>;<AK>;<BL>;<BS>;<HT>;/
          <LF>;<VT>;<FF>;<CR>;<SO>;<SI>;<DL>;<D1>;<D2>;<D3>;/
          <D4>;<NK>;<SY>;<EB>;<CN>;<EM>;<SB>;<EC>;<FS>;<GS>;/
          <RS>;<US>;<DT>;<PA>;<HO>;<BH>;<NH>;<IN>;<NL>;<SA>;/
          <ES>;<HS>;<HJ>;<VS>;<PD>;<PU>;<RI>;<S2>;<S3>;<DC>;/
          <P1>;<P2>;<TS>;<CC>;<MW>;<SG>;<EG>;<SS>;<GC>;<SC>;/
          <CI>;<ST>;<OC>;<PM>;<AC>

punct     <!>;<">;<Nb>;<DO>;<%>;<&>;<'>;<(>;<)>;<*>;/
          <+>;<,>;<->;<.>;<//>;<:>;<;>;<<>;<=>;</>>;/
          <?>;<At>;<<(>;<////>;<)/>>;<'/>>;<_>;<'!>;<(!>;<!!>;/
          <!)>;<'?>;<!I>;<Ct>;<Pd>;<Cu>;<Ye>;<BB>;<SE>;<':>;/
          <Co>;<-a>;<<<>;<NO>;<Rg>;<'->;<DG>;<+->;<2S>;<3S>;/
          <''>;<My>;<PI>;<.M>;<',>;<1S>;<-o>;</>/>>;<14>;<12>;/
          <34>;<?I>;<*X>;<-:>;<'6>;<"6>;<<->;<-!>;<-/>>;<-v>;/
          <'9>;<"9>;<'0>;<HB>;<TM>;<Md>;<18>;<38>;<58>;<78>;/
          <Om>;<'(>;<';>;<'<>;<'">;<'.>;<;S>;<Vs>;<1M>;<1N>;/
          <3M>;<4M>;<6M>;<1H>;<1T>;<-1>;<-N>;<-2>;<-M>;<-3>;/
          <'1>;<'2>;<'3>;<9'>;<9">;<.9>;<:9>;<<1>;</>1>;<<//>;/
          <///>>;<15>;<25>;<35>;<45>;<16>;<13>;<23>;<56>;<*->;/
          <//->;<//=>;<-X>;<%0>;<co>;<PO>;<Rx>;<AO>;<oC>;<Ml>;/
          <Fm>;<Tl>;<TR>;<MX>;<Mb>;<Mx>;<XX>;<OK>;<M2>;<!2>;/
          <=2>;<Ca>;<..>;<.3>;<:3>;<.:>;<:.>;<-+>;<!=>;<=3>;/
          <?1>;<?2>;<?->;<?=>;<=<>;</>=>;<0(>;<00>;<PP>;<-T>;/
          <-L>;<-V>;<AN>;<OR>;<.P>;<dP>;<f(>;<In>;<Io>;<RT>;/
          <*P>;<+Z>;<FA>;<TE>;<GF>;<DE>;<NB>;<(U>;<)U>;<(C>;/
          <)C>;<(_>;<)_>;<(->;<-)>;<</>>;<UD>;<Ub>;<<=>;<=/>>;/
          <==>;<//0>;<OL>;<0u>;<0U>;<SU>;<0:>;<OS>;<fS>;<Or>;/
          <SR>;<uT>;<UT>;<dT>;<Dt>;<PL>;<PR>;<*1>;<*2>;<VV>;/
          <HH>;<DR>;<LD>;<UR>;<UL>;<VR>;<VL>;<DH>;<UH>;<VH>;/
          <TB>;<LB>;<FB>;<sB>;<EH>;<vv>;<hh>;<dr>;<dl>;<ur>;/
          <ul>;<vr>;<vl>;<dh>;<uh>;<vh>;<.S>;<:S>;<?S>;<lB>;/
```

```
          <RB>;<cC>;<cD>;<Dr>;<Dl>;<Ur>;<Ul>;<Vr>;<Vl>;<dH>;/
          <uH>;<vH>;<Ob>;<Sb>;<Sn>;<Pt>;<NI>;<cH>;<cS>;<dR>;/
          <dL>;<uR>;<uL>;<vR>;<vL>;<Dh>;<Uh>;<Vh>;<0m>;<0M>;/
          <Ic>;<SM>;<CG>;<Ci>;<(A>;</>V>;<!<>;<<*>;<!/>>;<*/>>;/
          <<7>;<7<>;</>7>;<7/>>;<I2>;<0.>;<HI>;<::>;<FD>;<LZ>;/
          <BD>;<1R>;<2R>;<3R>;<4R>;<5R>;<6R>;<7R>;<8R>;<9R>;/
          <aR>;<bR>;<cR>;<N0>;<i3>;<;;>;<,,>;<!*>;<?*>;<;'>;/
          <,'>;<;!>;<,!>;<?;>;<?,>;<!:>;<?:>;<'%>;<,+>;<;+>;/
          <?+>;<++>;<:+>;<"+>;<=+>;<//+>;<'+>;<1+>;<3+>;<0+>;/
          <IS>;<,_>;<._>;<+">;<+_>;<*_>;<;_>;<0_>;<<+>;</>+>;/
          <<'>;</>'>;<<">;</>">;<(">;<)">;<=//>;<=_>;<('>;<)'>;/
          <KM>;<"5>;<05>;<*5>;<+5>;<-6>;<*6>;<+6>;<Iu>;<Il>;/
          <__>;<"!>;<"'>;<"/>>;<"?>;<"->;<"(>;<".>;<":>;<"//>;/          1
          <"0>;<",>;<"_>;<"">;<"<>;<";>;<"=>;<"1>;<"2>;<Fd>;/
          <Bd>;<Fl>;<Li>;<//f>;<0s>;<1s>;<2s>;<3s>;<4s>;<5s>;/
          <6s>;<7s>;<8s>;<9s>;<0S>;<4S>;<5S>;<6S>;<7S>;<8S>;/
          <9S>;<+S>;<-S>;<1h>;<2h>;<3h>;<4h>;<1j>;<2j>;<3j>;/
          <4j>;<UA>;<UB>;<yr>;<.6>;<<6>;</>6>;<,6>;<&6>;<(S>;/
          <)S>

tolower (<'A>,<'a>); (<'B>,<'b>); (<'D>,<'d>); (<'G>,<'g>); (<'J>,<'j>);/
          (<'Y>,<'y>); (<A>,<a>); (<A!>,<a!>); (<A'>,<a'>); (<A(>,<a(>);/          1
          (<A->,<a->); (<A1>,<a1>); (<A2>,<a2>); (<A3>,<a3>); (<A:>,<a:>);/
          (<A;>,<a;>); (<A<>,<a<>); (<A/>>,<a/>>); (<A?>,<a?>); (<AA>,<aa>);/
          (<AE>,<ae>); (<A_>,<a_>); (<B>,<b>); (<B.>,<b.>); (<B_>,<b_>);/
          (<C>,<c>); (<C'>,<c'>); (<C,>,<c,>); (<C.>,<c.>); (<C<>,<c<>);/
          (<C/>>,<c/>>); (<D>,<d>); (<D->,<d->); (<D.>,<d.>); (<D//>,<d//>);/
          (<D;>,<d;>); (<D<>,<d<>); (<D_>,<d_>); (<E>,<e>); (<E!>,<e!>);/
          (<E'>,<e'>); (<E(>,<e(>); (<E->,<e->); (<E.>,<e.>); (<E:>,<e:>);/
          (<E;>,<e;>); (<E<>,<e<>); (<E/>>,<e/>>); (<E?>,<e?>); (<ED>,<ed>);/
          (<EZ>,<ez>); (<E_>,<e_>); (<F>,<f>); (<F.>,<f.>);/
          (<G>,<ft>); (<G'>,<g'>); (<G(>,<g(>); (<G,>,<g,>);/
          (<G->,<g->); (<G.>,<g.>); (<G//>,<g//>); (<G<>,<g<>); (<G/>>,<g/>>);/
          (<H>,<h>); (<H,>,<h,>); (<H.>,<h.>); (<H//>,<h//>); (<H:>,<h:>);/
          (<H;>,<h;>); (<H/>>,<h/>>); (<I>,<i>); (<I!>,<i!>); (<I'>,<i'>);/
          (<I(>,<i(>); (<I->,<i->); (<I.>,<i.>); (<I:>,<i:>); (<I;>,<i;>);/
          (<I<>,<i<>); (<I/>>,<i/>>); (<I?>,<i?>); (<IJ>,<ij>); (<J>,<j>);/
          (<J(>,<j(>); (<J/>>,<j/>>); (<K>,<k>); (<K'>,<k'>); (<K,>,<k,>);/
          (<K.>,<k.>); (<K;>,<k;>); (<K<>,<k<>); (<K_>,<k_>);/
          (<L>,<l>); (<L'>,<l'>); (<L,>,<l,>); (<L.>,<l.>); (<L//>,<l//>);/
          (<L<>,<l<>); (<L_>,<l_>); (<M>,<m>); (<M'>,<m'>); (<M.>,<m.>);/
          (<N>,<n>); (<N'>,<n'>); (<N,>,<n,>); (<N.>,<n.>); (<N<>,<n<>);/
          (<N?>,<n?>); (<NG>,<ng>); (<N_>,<n_>); (<O>,<o>); (<O!>,<o!>);/
          (<O">,<o">); (<O'>,<o'>); (<O(>,<o(>); (<O->,<o->); (<O//>,<o//>);/
          (<O1>,<o1>); (<O:>,<o:>); (<O;>,<o;>); (<O<>,<o<>); (<O/>>,<o/>>);/
          (<O?>,<o?>); (<OE>,<oe>); (<O_>,<o_>); (<P>,<p>); (<P'>,<p'>);/
          (<Q>,<q>); (<R>,<r>); (<R'>,<r'>); (<R,>,<r,>); (<R.>,<r.>);/
          (<R<>,<r<>); (<R_>,<r_>); (<S>,<s>); (<S'>,<s'>); (<S,>,<s,>);/
          (<S.>,<s.>); (<S;>,<s;>); (<S<>,<s<>); (<S/>>,<s/>>); (<T>,<st>);/
          (<T,>,<t>); (<T.>,<t.>); (<T//>,<t//>); (<T<>,<t<>);/
          (<TH>,<th>); (<T_>,<t_>); (<U>,<u>); (<U!>,<u!>); (<U">,<u">);/
          (<U'>,<u'>); (<U(>,<u(>); (<U->,<u->); (<U0>,<u0>); (<U:>,<u:>);/          1
          (<U;>,<u;>); (<U<>,<u<>); (<U/>>,<u/>>); (<U?>,<u?>); (<V>,<v>);/
          (<V?>,<v?>); (<W>,<w>); (<W'>,<w'>); (<W.>,<w.>); (<W:>,<w:>);/
          (<W/>>,<w/>>); (<X>,<x>); (<X.>,<x.>); (<X:>,<x:>); (<Y>,<y>);/
          (<Y!>,<y!>); (<Y'>,<y'>); (<Y.>,<y.>); (<Y:>,<y:>); (<Y/>>,<y/>>);/
          (<Z>,<z>); (<Z'>,<z'>); (<Z(>,<z(>); (<Z.>,<z.>); (<Z//>,<z//>);/
```

```
          (<Z<>,<z<>); (<Z/>>,<z/>>); (<Z_>,<z_>); (<%">,<%’>); (<=">,<=’>);/
          (<A=>,<a=>); (<B=>,<b=>); (<C%>,<c%>); (<C=>,<c=>); (<D%>,<d%>);/
          (<D=>,<d=>); (<DS>,<ds>); (<DZ>,<dz>); (<E=>,<e=>); (<F=>,<f=>);/
          (<G%>,<g%>); (<G=>,<g=>); (<H=>,<h=>); (<I=>,<i=>); (<IE>,<ie>);/
          (<II>,<ii>); (<IO>,<io>); (<J%>,<j%>); (<J=>,<j=>); (<JA>,<ja>);/
          (<JE>,<je>); (<JU>,<ju>); (<K=>,<k=>); (<KJ>,<kj>); (<L=>,<l=>);/
          (<LJ>,<lj>); (<M=>,<m=>); (<N=>,<n=>); (<NJ>,<nj>); (<O=>,<o=>);/
          (<P=>,<p=>); (<R=>,<r=>); (<S%>,<s%>); (<S=>,<s=>); (<Sc>,<sc>);/
          (<T=>,<t=>); (<Ts>,<ts>); (<U=>,<u=>); (<V=>,<v=>); (<Y=>,<y=>);/
          (<YI>,<yi>); (<Z%>,<z%>); (<Z=>,<z=>); (<A%>,<a%>); (<A*>,<a*>);/
          (<B*>,<b*>); (<C*>,<c*>); (<D*>,<d*>); (<E%>,<e%>); (<E*>,<e*>);/
          (<F*>,<f*>); (<G*>,<g*>); (<H*>,<h*>); (<I%>,<i%>); (<I*>,<i*>);/
          (<J*>,<j*>); (<K*>,<k*>); (<L*>,<l*>); (<M*>,<m*>); (<N*>,<n*>);/
          (<O%>,<o%>); (<O*>,<o*>); (<P*>,<p*>); (<Q*>,<q*>); (<R*>,<r*>);/
          (<S*>,<s*>); (<T*>,<t*>); (<U%>,<u%>); (<U*>,<u*>); (<V*>,<v*>);/
          (<W%>,<w%>); (<W*>,<w*>); (<X*>,<x*>); (<Y%>,<y%>); (<Y*>,<y*>);/
          (<Z*>,<z*>)

toupper (<’a>,<’A>); (<’b>,<’B>); (<’d>,<’D>); (<’g>,<’G>); (<’j>,<’J>);/
          (<’y>,<’Y>); (<a>,<A>); (<a!>,<A!>); (<a’>,<A’>); (<a(>,<A(>);/        1
          (<a->,<A->); (<a1>,<A1>); (<a2>,<A2>); (<a3>,<A3>); (<a:>,<A:>);/
          (<a;>,<A;>); (<a<>,<A<>); (<a/>>,<A/>>); (<a?>,<A?>); (<aa>,<AA>);/
          (<ae>,<AE>); (<a_>,<A_>); (<b>,<B>); (<b.>,<B.>); (<b_>,<B_>);/
          (<c>,<C>); (<c’>,<C’>); (<c,>,<C,>); (<c.>,<C.>); (<c<>,<C<>);/
          (<c/>>,<C/>>); (<d>,<D>); (<d->,<D->); (<d.>,<D.>); (<d//>,<D//>);/
          (<d;>,<D;>); (<d<>,<D<>); (<d_>,<D_>); (<e>,<E>); (<e!>,<E!>);/
          (<e’>,<E’>); (<e(>,<E(>); (<e->,<E->); (<e.>,<E.>); (<e:>,<E:>);/
          (<e;>,<E;>); (<e<>,<E<>); (<e/>>,<E/>>); (<e?>,<E?>); (<ed>,<ED>);/
          (<ez>,<EZ>); (<e_>,<E_>); (<f>,<F>); (<f.>,<F.>);/
          (<ft>,<G>); (<g’>,<G’>); (<g(>,<G(>); (<g,>,<G,>);/
          (<g->,<G->); (<g.>,<G.>); (<g//>,<G//>); (<g<>,<G<>); (<g/>>,<G/>>);/
          (<h>,<H>); (<h,>,<H,>); (<h.>,<H.>); (<h//>,<H//>); (<h:>,<H:>);/
          (<h;>,<H;>); (<h/>>,<H/>>); (<i>,<I>); (<i!>,<I!>); (<i’>,<I’>);/
          (<i(>,<I(>); (<i->,<I->); (<i.>,<I.>); (<i:>,<I:>); (<i;>,<I;>);/
          (<i<>,<I<>); (<i/>>,<I/>>); (<i?>,<I?>); (<ij>,<IJ>); (<j>,<J>);/
          (<j(>,<J(>); (<j/>>,<J/>>); (<k>,<K>); (<k’>,<K’>); (<k,>,<K,>);/
          (<k.>,<K.>); (<k;>,<K;>); (<k<>,<K<>); (<k_>,<K_>);/
          (<l>,<L>); (<l’>,<L’>); (<l,>,<L,>); (<l.>,<L.>); (<l//>,<L//>);/
          (<l<>,<L<>); (<l_>,<L_>); (<m>,<M>); (<m’>,<M’>); (<m.>,<M.>);/
          (<n>,<N>); (<n’>,<N’>); (<n,>,<N,>); (<n.>,<N.>); (<n<>,<N<>);/
          (<n?>,<N?>); (<ng>,<NG>); (<n_>,<N_>); (<o>,<O>); (<o!>,<O!>);/
          (<o">,<O">); (<o’>,<O’>); (<o(>,<O(>); (<o->,<O->); (<o//>,<O//>);/
          (<o1>,<O1>); (<o:>,<O:>); (<o;>,<O;>); (<o<>,<O<>); (<o/>>,<O/>>);/
          (<o?>,<O?>); (<oe>,<OE>); (<o_>,<O_>); (<p>,<P>); (<p’>,<P’>);/
          (<q>,<Q>); (<r>,<R>); (<r’>,<R’>); (<r,>,<R,>); (<r.>,<R.>);/
          (<r<>,<R<>); (<r_>,<R_>); (<s>,<S>); (<s’>,<S’>); (<s,>,<S,>);/
          (<s.>,<S.>); (<s;>,<S;>); (<s<>,<S<>); (<s/>>,<S/>>); (<st>,<T>);/
          (<t>,<T,>); (<t.>,<T.>); (<t//>,<T//>); (<t<>,<T<>);/
          (<th>,<TH>); (<t_>,<T_>); (<u>,<U>); (<u!>,<U!>); (<u">,<U">);/
          (<u’>,<U’>); (<u(>,<U(>); (<u->,<U->); (<u0>,<U0>); (<u:>,<U:>);/      1
          (<u;>,<U;>); (<u<>,<U<>); (<u/>>,<U/>>); (<u?>,<U?>); (<v>,<V>);/
          (<v?>,<V?>); (<w>,<W>); (<w’>,<W’>); (<w.>,<W.>); (<w:>,<W:>);/
          (<w/>>,<W/>>); (<x>,<X>); (<x.>,<X.>); (<x:>,<X:>); (<y>,<Y>);/
          (<y!>,<Y!>); (<y’>,<Y’>); (<y.>,<Y.>); (<y:>,<Y:>); (<y/>>,<Y/>>);/
          (<z>,<Z>); (<z’>,<Z’>); (<z(>,<Z(>); (<z.>,<Z.>); (<z//>,<Z//>);/
          (<z<>,<Z<>); (<z/>>,<Z/>>); (<z_>,<Z_>); (<%’>,<%">); (<=’>,<=">);/
          (<a=>,<A=>); (<b=>,<B=>); (<c%>,<C%>); (<c=>,<C=>); (<d%>,<D%>);/
```

```
        (<d=>,<D=>); (<ds>,<DS>); (<dz>,<DZ>); (<e=>,<E=>); (<f=>,<F=>);/
        (<g%>,<G%>); (<g=>,<G=>); (<h=>,<H=>); (<i=>,<I=>); (<ie>,<IE>);/
        (<ii>,<II>); (<io>,<IO>); (<j%>,<J%>); (<j=>,<J=>); (<ja>,<JA>);/
        (<je>,<JE>); (<ju>,<JU>); (<k=>,<K=>); (<kj>,<KJ>); (<l=>,<L=>);/
        (<lj>,<LJ>); (<m=>,<M=>); (<n=>,<N=>); (<nj>,<NJ>); (<o=>,<O=>);/
        (<p=>,<P=>); (<r=>,<R=>); (<s%>,<S%>); (<s=>,<S=>); (<sc>,<Sc>);/
        (<t=>,<T=>); (<ts>,<Ts>); (<u=>,<U=>); (<v=>,<V=>); (<y=>,<Y=>);/
        (<yi>,<YI>); (<z%>,<Z%>); (<z=>,<Z=>); (<a%>,<A%>); (<a*>,<A*>);/
        (<b*>,<B*>); (<c*>,<C*>); (<d*>,<D*>); (<e%>,<E%>); (<e*>,<E*>);/
        (<f*>,<F*>); (<g*>,<G*>); (<h*>,<H*>); (<i%>,<I%>); (<i*>,<I*>);/
        (<j*>,<J*>); (<k*>,<K*>); (<l*>,<L*>); (<m*>,<M*>); (<n*>,<N*>);/
        (<o%>,<O%>); (<o*>,<O*>); (<p*>,<P*>); (<q*>,<Q*>); (<r*>,<R*>);/
        (<s*>,<S*>); (<*s>,<S*>); (<t*>,<T*>); (<u%>,<U%>); (<u*>,<U*>);/     1
        (<v*>,<V*>); (<w%>,<W%>); (<w*>,<W*>); (<x*>,<X*>); (<y%>,<Y%>);/     1
        (<y*>,<Y*>); (<z*>,<Z*>)                                              1

END LC_CTYPE

LC_COLLATE

% Ordering algorithm:                                                         1
%  1. Spaces and hyphen (but not soft hyphen) before punctuation              1
%     characters, punctuation characters before numbers,                      1
%     numbers before letters.                                                 1
%  2. Letters with diacritical marks are members of equivalence classes       1
%  3. Upper case letters before corresponding lower case letter.              1
%  4. Specials are ignored when comparing letters, but then they are considered 1
%  5. The alphabets are sorted in the order of appearance in ISO 10646:       1
%     Latin, Cyrillic, Greek, Arabic and Hebrew.                              1
%  6. In Danish, the letter combination ‘aa’ is equivalent to ‘<aa>’          1
%                                                                             1
% The ordering algorithm is in accordance with Danish Standard DS 377         1
% and the Danish Orthography Dictionary (Retskrivningsordbogen, 1986).        1
% It is also in accordance with Faroese and Greenlandic orthography.          1

collating-element <A-A> from <A><A>                                           1
collating-element <a-a> from <a><a>                                           1
collating-element <A-a> from <A><a>                                           1
collating-element <s-s> from <s><s>                                           1
collating-element <i-j> from <i><j>                                           1
collating-element <I-J> from <I><J>                                           1
collating-element <o-e> from <o><e>                                           1
collating-element <O-E> from <O><E>                                           1
collating-element <t-h> from <t><h>                                           1
collating-element <T-H> from <T><H>                                           1
collating-element <n-g> from <n><g>                                           1
collating-element <N-G> from <N><G>                                           1

% collating symbols, <CAPITAL> or <SMALL> letters first                       1
% <CAPITAL> letters before <SMALL> letters                                    1

collating-symbol <CAPITAL>
collating-symbol <BOTH>
collating-symbol <SMALL>

collating-symbol <NO-ACCENT>
collating-symbol <ACUTE>
```

```
collating-symbol <GRAVE>
collating-symbol <CIRCUMFLEX>
collating-symbol <TILDE>
collating-symbol <MACRON>
collating-symbol <BREVE>
collating-symbol <DOT>
collating-symbol <DIAERESIS>
collating-symbol <CEDILLA>
collating-symbol <UNDERLINE>
collating-symbol <STROKE>
collating-symbol <DOUBLE-ACUTE>
collating-symbol <OGONEK>
collating-symbol <CARON>
collating-symbol <CYRILLIC>
collating-symbol <GREEK>
collating-symbol <ALPHA-1>
collating-symbol <ALPHA-2>
collating-symbol <PRECEDED-BY-APOSTROPHE>
collating-symbol <SPECIAL>
collating-symbol <ACC0>
collating-symbol <ACC1>
collating-symbol <ACC2>
collating-symbol <ACC3>
collating-symbol <ACC11>
collating-symbol <ACC12>

% letter;accent;case;specials                                          1

order_start forward;backward;forward;forward

<CAPITAL>
<BOTH>
<SMALL>

<NO-ACCENT>
<ACUTE>
<GRAVE>
<CIRCUMFLEX>
<TILDE>
<MACRON>
<BREVE>
<DOT>
<DIAERESIS>
<CEDILLA>
<UNDERLINE>
<STROKE>
<DOUBLE-ACUTE>
<OGONEK>
<CARON>
<CYRILLIC>
<GREEK>
<ALPHA-1>
<ALPHA-2>
<PRECEDED-BY-APOSTROPHE>
<SPECIAL>
<ACC0>
<ACC1>
```

```
<ACC2>
<ACC3>
<ACC11>
<ACC12>

<SP>    <SP>
<NS>    <SP>
<HT>    <SP>
<VT>    <SP>
<CR>    <SP>
<LF>    <SP>
<FF>    <SP>
<->     <SP>
<//>    <SP>
<!>     IGNORE;IGNORE;IGNORE
<">     IGNORE;IGNORE;IGNORE
<Nb>    IGNORE;IGNORE;IGNORE
<DO>    IGNORE;IGNORE;IGNORE
<%>     IGNORE;IGNORE;IGNORE
<&>     IGNORE;IGNORE;IGNORE
<'>     IGNORE;IGNORE;IGNORE
<(>     IGNORE;IGNORE;IGNORE
<)>     IGNORE;IGNORE;IGNORE
<*>     IGNORE;IGNORE;IGNORE
<+>     IGNORE;IGNORE;IGNORE
<,>     IGNORE;IGNORE;IGNORE
<.>     IGNORE;IGNORE;IGNORE
<:>     IGNORE;IGNORE;IGNORE
<;>     IGNORE;IGNORE;IGNORE
<<>     IGNORE;IGNORE;IGNORE
<=>     IGNORE;IGNORE;IGNORE
</>>    IGNORE;IGNORE;IGNORE
<?>     IGNORE;IGNORE;IGNORE
<At>    IGNORE;IGNORE;IGNORE
<<(>    IGNORE;IGNORE;IGNORE
<////>  IGNORE;IGNORE;IGNORE
<)/>>   IGNORE;IGNORE;IGNORE
<'/>>   IGNORE;IGNORE;IGNORE
<_>     IGNORE;IGNORE;IGNORE
<'!>    IGNORE;IGNORE;IGNORE
<(!>    IGNORE;IGNORE;IGNORE
<!!>    IGNORE;IGNORE;IGNORE
<!)>    IGNORE;IGNORE;IGNORE
<'?>    IGNORE;IGNORE;IGNORE
<!I>    IGNORE;IGNORE;IGNORE
<Ct>    IGNORE;IGNORE;IGNORE
<Pd>    IGNORE;IGNORE;IGNORE
<Cu>    IGNORE;IGNORE;IGNORE
<Ye>    IGNORE;IGNORE;IGNORE
<BB>    IGNORE;IGNORE;IGNORE
<SE>    IGNORE;IGNORE;IGNORE
<':>    IGNORE;IGNORE;IGNORE
<Co>    IGNORE;IGNORE;IGNORE
<-a>    IGNORE;IGNORE;IGNORE
<<<>    IGNORE;IGNORE;IGNORE
<NO>    IGNORE;IGNORE;IGNORE
<Rg>    IGNORE;IGNORE;IGNORE
```

```
<'->     IGNORE;IGNORE;IGNORE
<DG>     IGNORE;IGNORE;IGNORE
<+->     IGNORE;IGNORE;IGNORE
<''>     IGNORE;IGNORE;IGNORE
<My>     IGNORE;IGNORE;IGNORE
<PI>     IGNORE;IGNORE;IGNORE
<.M>     IGNORE;IGNORE;IGNORE
<',>     IGNORE;IGNORE;IGNORE
<-o>     IGNORE;IGNORE;IGNORE
</>/>>   IGNORE;IGNORE;IGNORE
<14>     IGNORE;IGNORE;IGNORE
<12>     IGNORE;IGNORE;IGNORE
<34>     IGNORE;IGNORE;IGNORE
<?I>     IGNORE;IGNORE;IGNORE
<*X>     IGNORE;IGNORE;IGNORE
<-:>     IGNORE;IGNORE;IGNORE
<'6>     IGNORE;IGNORE;IGNORE
<"6>     IGNORE;IGNORE;IGNORE
<-!>     IGNORE;IGNORE;IGNORE
<-v>     IGNORE;IGNORE;IGNORE
<'9>     IGNORE;IGNORE;IGNORE
<"9>     IGNORE;IGNORE;IGNORE
<'0>     IGNORE;IGNORE;IGNORE
<HB>     IGNORE;IGNORE;IGNORE
<TM>     IGNORE;IGNORE;IGNORE
<Md>     IGNORE;IGNORE;IGNORE
<18>     IGNORE;IGNORE;IGNORE
<38>     IGNORE;IGNORE;IGNORE
<58>     IGNORE;IGNORE;IGNORE
<78>     IGNORE;IGNORE;IGNORE
<Om>     IGNORE;IGNORE;IGNORE
<'(>     IGNORE;IGNORE;IGNORE
<';>     IGNORE;IGNORE;IGNORE
<'<>     IGNORE;IGNORE;IGNORE
<'">     IGNORE;IGNORE;IGNORE
<'.>     IGNORE;IGNORE;IGNORE
<;S>     IGNORE;IGNORE;IGNORE
<Vs>     IGNORE;IGNORE;IGNORE
<1M>     IGNORE;IGNORE;IGNORE
<1N>     IGNORE;IGNORE;IGNORE
<3M>     IGNORE;IGNORE;IGNORE
<4M>     IGNORE;IGNORE;IGNORE
<6M>     IGNORE;IGNORE;IGNORE
<1H>     IGNORE;IGNORE;IGNORE
<1T>     IGNORE;IGNORE;IGNORE
<-1>     IGNORE;IGNORE;IGNORE
<-N>     IGNORE;IGNORE;IGNORE
<-2>     IGNORE;IGNORE;IGNORE
<-M>     IGNORE;IGNORE;IGNORE
<-3>     IGNORE;IGNORE;IGNORE
<'1>     IGNORE;IGNORE;IGNORE
<'2>     IGNORE;IGNORE;IGNORE
<'3>     IGNORE;IGNORE;IGNORE
<9'>     IGNORE;IGNORE;IGNORE
<9">     IGNORE;IGNORE;IGNORE
<.9>     IGNORE;IGNORE;IGNORE
<:9>     IGNORE;IGNORE;IGNORE
```

```
<<1>     IGNORE;IGNORE;IGNORE
</>1>    IGNORE;IGNORE;IGNORE
<15>     IGNORE;IGNORE;IGNORE
<25>     IGNORE;IGNORE;IGNORE
<35>     IGNORE;IGNORE;IGNORE
<45>     IGNORE;IGNORE;IGNORE
<16>     IGNORE;IGNORE;IGNORE
<13>     IGNORE;IGNORE;IGNORE
<23>     IGNORE;IGNORE;IGNORE
<56>     IGNORE;IGNORE;IGNORE
<*->     IGNORE;IGNORE;IGNORE
<//->    IGNORE;IGNORE;IGNORE
<//=>    IGNORE;IGNORE;IGNORE
<-X>     IGNORE;IGNORE;IGNORE
<%0>     IGNORE;IGNORE;IGNORE
<co>     IGNORE;IGNORE;IGNORE
<PO>     IGNORE;IGNORE;IGNORE
<Rx>     IGNORE;IGNORE;IGNORE
<AO>     IGNORE;IGNORE;IGNORE
<oC>     IGNORE;IGNORE;IGNORE
<Ml>     IGNORE;IGNORE;IGNORE
<Fm>     IGNORE;IGNORE;IGNORE
<Tl>     IGNORE;IGNORE;IGNORE
<TR>     IGNORE;IGNORE;IGNORE
<MX>     IGNORE;IGNORE;IGNORE
<Mb>     IGNORE;IGNORE;IGNORE
<Mx>     IGNORE;IGNORE;IGNORE
<XX>     IGNORE;IGNORE;IGNORE
<OK>     IGNORE;IGNORE;IGNORE
<M2>     IGNORE;IGNORE;IGNORE
<!2>     IGNORE;IGNORE;IGNORE
<=2>     IGNORE;IGNORE;IGNORE
<Ca>     IGNORE;IGNORE;IGNORE
<..>     IGNORE;IGNORE;IGNORE
<.3>     IGNORE;IGNORE;IGNORE
<:3>     IGNORE;IGNORE;IGNORE
<.:>     IGNORE;IGNORE;IGNORE
<:.>     IGNORE;IGNORE;IGNORE
<-+>     IGNORE;IGNORE;IGNORE
<!=>     IGNORE;IGNORE;IGNORE
<=3>     IGNORE;IGNORE;IGNORE
<?1>     IGNORE;IGNORE;IGNORE
<?2>     IGNORE;IGNORE;IGNORE
<?->     IGNORE;IGNORE;IGNORE
<?=>     IGNORE;IGNORE;IGNORE
<=<>     IGNORE;IGNORE;IGNORE
</>=>    IGNORE;IGNORE;IGNORE
<0(>     IGNORE;IGNORE;IGNORE
<00>     IGNORE;IGNORE;IGNORE
<PP>     IGNORE;IGNORE;IGNORE
<-T>     IGNORE;IGNORE;IGNORE
<-L>     IGNORE;IGNORE;IGNORE
<-V>     IGNORE;IGNORE;IGNORE
<AN>     IGNORE;IGNORE;IGNORE
<OR>     IGNORE;IGNORE;IGNORE
<.P>     IGNORE;IGNORE;IGNORE
<dP>     IGNORE;IGNORE;IGNORE
```

```
<f(>    IGNORE;IGNORE;IGNORE
<In>    IGNORE;IGNORE;IGNORE
<Io>    IGNORE;IGNORE;IGNORE
<RT>    IGNORE;IGNORE;IGNORE
<*P>    IGNORE;IGNORE;IGNORE
<+Z>    IGNORE;IGNORE;IGNORE
<FA>    IGNORE;IGNORE;IGNORE
<TE>    IGNORE;IGNORE;IGNORE
<GF>    IGNORE;IGNORE;IGNORE
<DE>    IGNORE;IGNORE;IGNORE
<NB>    IGNORE;IGNORE;IGNORE
<(U>    IGNORE;IGNORE;IGNORE
<)U>    IGNORE;IGNORE;IGNORE
<(C>    IGNORE;IGNORE;IGNORE
<)C>    IGNORE;IGNORE;IGNORE
<(_>    IGNORE;IGNORE;IGNORE
<)_>    IGNORE;IGNORE;IGNORE
<(->    IGNORE;IGNORE;IGNORE
<-)>    IGNORE;IGNORE;IGNORE
<</>>   IGNORE;IGNORE;IGNORE
<UD>    IGNORE;IGNORE;IGNORE
<Ub>    IGNORE;IGNORE;IGNORE
<<=>    IGNORE;IGNORE;IGNORE
<=/>>   IGNORE;IGNORE;IGNORE
<==>    IGNORE;IGNORE;IGNORE
<//0>   IGNORE;IGNORE;IGNORE
<OL>    IGNORE;IGNORE;IGNORE
<0u>    IGNORE;IGNORE;IGNORE
<0U>    IGNORE;IGNORE;IGNORE
<SU>    IGNORE;IGNORE;IGNORE
<0:>    IGNORE;IGNORE;IGNORE
<OS>    IGNORE;IGNORE;IGNORE
<fS>    IGNORE;IGNORE;IGNORE
<Or>    IGNORE;IGNORE;IGNORE
<SR>    IGNORE;IGNORE;IGNORE
<uT>    IGNORE;IGNORE;IGNORE
<UT>    IGNORE;IGNORE;IGNORE
<dT>    IGNORE;IGNORE;IGNORE
<Dt>    IGNORE;IGNORE;IGNORE
<PL>    IGNORE;IGNORE;IGNORE
<PR>    IGNORE;IGNORE;IGNORE
<*1>    IGNORE;IGNORE;IGNORE
<*2>    IGNORE;IGNORE;IGNORE
<VV>    IGNORE;IGNORE;IGNORE
<HH>    IGNORE;IGNORE;IGNORE
<DR>    IGNORE;IGNORE;IGNORE
<LD>    IGNORE;IGNORE;IGNORE
<UR>    IGNORE;IGNORE;IGNORE
<UL>    IGNORE;IGNORE;IGNORE
<VR>    IGNORE;IGNORE;IGNORE
<VL>    IGNORE;IGNORE;IGNORE
<DH>    IGNORE;IGNORE;IGNORE
<UH>    IGNORE;IGNORE;IGNORE
<VH>    IGNORE;IGNORE;IGNORE
<TB>    IGNORE;IGNORE;IGNORE
<LB>    IGNORE;IGNORE;IGNORE
<FB>    IGNORE;IGNORE;IGNORE
```

```
<sB>    IGNORE;IGNORE;IGNORE
<EH>    IGNORE;IGNORE;IGNORE
<vv>    IGNORE;IGNORE;IGNORE
<hh>    IGNORE;IGNORE;IGNORE
<dr>    IGNORE;IGNORE;IGNORE
<dl>    IGNORE;IGNORE;IGNORE
<ur>    IGNORE;IGNORE;IGNORE
<ul>    IGNORE;IGNORE;IGNORE
<vr>    IGNORE;IGNORE;IGNORE
<vl>    IGNORE;IGNORE;IGNORE
<dh>    IGNORE;IGNORE;IGNORE
<uh>    IGNORE;IGNORE;IGNORE
<vh>    IGNORE;IGNORE;IGNORE
<.S>    IGNORE;IGNORE;IGNORE
<:S>    IGNORE;IGNORE;IGNORE
<?S>    IGNORE;IGNORE;IGNORE
<lB>    IGNORE;IGNORE;IGNORE
<RB>    IGNORE;IGNORE;IGNORE
<cC>    IGNORE;IGNORE;IGNORE
<cD>    IGNORE;IGNORE;IGNORE
<Dr>    IGNORE;IGNORE;IGNORE
<Dl>    IGNORE;IGNORE;IGNORE
<Ur>    IGNORE;IGNORE;IGNORE
<Ul>    IGNORE;IGNORE;IGNORE
<Vr>    IGNORE;IGNORE;IGNORE
<Vl>    IGNORE;IGNORE;IGNORE
<dH>    IGNORE;IGNORE;IGNORE
<uH>    IGNORE;IGNORE;IGNORE
<vH>    IGNORE;IGNORE;IGNORE
<Ob>    IGNORE;IGNORE;IGNORE
<Sb>    IGNORE;IGNORE;IGNORE
<Sn>    IGNORE;IGNORE;IGNORE
<Pt>    IGNORE;IGNORE;IGNORE
<NI>    IGNORE;IGNORE;IGNORE
<cH>    IGNORE;IGNORE;IGNORE
<cS>    IGNORE;IGNORE;IGNORE
<dR>    IGNORE;IGNORE;IGNORE
<dL>    IGNORE;IGNORE;IGNORE
<uR>    IGNORE;IGNORE;IGNORE
<uL>    IGNORE;IGNORE;IGNORE
<vR>    IGNORE;IGNORE;IGNORE
<vL>    IGNORE;IGNORE;IGNORE
<Dh>    IGNORE;IGNORE;IGNORE
<Uh>    IGNORE;IGNORE;IGNORE
<Vh>    IGNORE;IGNORE;IGNORE
<0m>    IGNORE;IGNORE;IGNORE
<0M>    IGNORE;IGNORE;IGNORE
<Ic>    IGNORE;IGNORE;IGNORE
<SM>    IGNORE;IGNORE;IGNORE
<CG>    IGNORE;IGNORE;IGNORE
<Ci>    IGNORE;IGNORE;IGNORE
<(A>    IGNORE;IGNORE;IGNORE
</>V>   IGNORE;IGNORE;IGNORE
<!<>    IGNORE;IGNORE;IGNORE
<<*>    IGNORE;IGNORE;IGNORE
<!/>>   IGNORE;IGNORE;IGNORE
<*/>>   IGNORE;IGNORE;IGNORE
```

```
<<7>    IGNORE;IGNORE;IGNORE
<7<>    IGNORE;IGNORE;IGNORE
</>7>   IGNORE;IGNORE;IGNORE
<7/>>   IGNORE;IGNORE;IGNORE
<I2>    IGNORE;IGNORE;IGNORE
<0.>    IGNORE;IGNORE;IGNORE
<HI>    IGNORE;IGNORE;IGNORE
<::>    IGNORE;IGNORE;IGNORE
<FD>    IGNORE;IGNORE;IGNORE
<LZ>    IGNORE;IGNORE;IGNORE
<BD>    IGNORE;IGNORE;IGNORE
<1R>    IGNORE;IGNORE;IGNORE
<2R>    IGNORE;IGNORE;IGNORE
<3R>    IGNORE;IGNORE;IGNORE
<4R>    IGNORE;IGNORE;IGNORE
<5R>    IGNORE;IGNORE;IGNORE
<6R>    IGNORE;IGNORE;IGNORE
<7R>    IGNORE;IGNORE;IGNORE
<8R>    IGNORE;IGNORE;IGNORE
<9R>    IGNORE;IGNORE;IGNORE
<aR>    IGNORE;IGNORE;IGNORE
<bR>    IGNORE;IGNORE;IGNORE
<cR>    IGNORE;IGNORE;IGNORE
<N0>    IGNORE;IGNORE;IGNORE
<i3>    IGNORE;IGNORE;IGNORE
<;;>    IGNORE;IGNORE;IGNORE
<,,>    IGNORE;IGNORE;IGNORE
<!*>    IGNORE;IGNORE;IGNORE
<?*>    IGNORE;IGNORE;IGNORE
<;'>    IGNORE;IGNORE;IGNORE
<,'>    IGNORE;IGNORE;IGNORE
<;!>    IGNORE;IGNORE;IGNORE
<,!>    IGNORE;IGNORE;IGNORE
<?;>    IGNORE;IGNORE;IGNORE
<?,>    IGNORE;IGNORE;IGNORE
<!:>    IGNORE;IGNORE;IGNORE
<?:>    IGNORE;IGNORE;IGNORE
<'%>    IGNORE;IGNORE;IGNORE
<,+>    IGNORE;IGNORE;IGNORE
<;+>    IGNORE;IGNORE;IGNORE
<?+>    IGNORE;IGNORE;IGNORE
<++>    IGNORE;IGNORE;IGNORE
<:+>    IGNORE;IGNORE;IGNORE
<"+>    IGNORE;IGNORE;IGNORE
<=+>    IGNORE;IGNORE;IGNORE
<//+>   IGNORE;IGNORE;IGNORE
<'+>    IGNORE;IGNORE;IGNORE
<1+>    IGNORE;IGNORE;IGNORE
<3+>    IGNORE;IGNORE;IGNORE
<0+>    IGNORE;IGNORE;IGNORE
<IS>    IGNORE;IGNORE;IGNORE
<,_>    IGNORE;IGNORE;IGNORE
<._>    IGNORE;IGNORE;IGNORE
<+">    IGNORE;IGNORE;IGNORE
<+_>    IGNORE;IGNORE;IGNORE
<*_>    IGNORE;IGNORE;IGNORE
<;_>    IGNORE;IGNORE;IGNORE
```

```
<0_>     IGNORE;IGNORE;IGNORE
<<+>     IGNORE;IGNORE;IGNORE
</>+>    IGNORE;IGNORE;IGNORE
<<'>     IGNORE;IGNORE;IGNORE
</>'>    IGNORE;IGNORE;IGNORE
<<">     IGNORE;IGNORE;IGNORE
</>">    IGNORE;IGNORE;IGNORE
<(">     IGNORE;IGNORE;IGNORE
<)">     IGNORE;IGNORE;IGNORE
<=//>    IGNORE;IGNORE;IGNORE
<=_>     IGNORE;IGNORE;IGNORE
<('>     IGNORE;IGNORE;IGNORE
<)'>     IGNORE;IGNORE;IGNORE
<KM>     IGNORE;IGNORE;IGNORE
<"5>     IGNORE;IGNORE;IGNORE
<05>     IGNORE;IGNORE;IGNORE
<*5>     IGNORE;IGNORE;IGNORE
<+5>     IGNORE;IGNORE;IGNORE
<-6>     IGNORE;IGNORE;IGNORE
<*6>     IGNORE;IGNORE;IGNORE
<+6>     IGNORE;IGNORE;IGNORE
<Iu>     IGNORE;IGNORE;IGNORE
<Il>     IGNORE;IGNORE;IGNORE
<NU>     IGNORE;IGNORE;IGNORE
<SH>     IGNORE;IGNORE;IGNORE
<SX>     IGNORE;IGNORE;IGNORE
<EX>     IGNORE;IGNORE;IGNORE
<ET>     IGNORE;IGNORE;IGNORE
<EQ>     IGNORE;IGNORE;IGNORE
<AK>     IGNORE;IGNORE;IGNORE
<BL>     IGNORE;IGNORE;IGNORE
<BS>     IGNORE;IGNORE;IGNORE
<SO>     IGNORE;IGNORE;IGNORE
<SI>     IGNORE;IGNORE;IGNORE
<DL>     IGNORE;IGNORE;IGNORE
<D1>     IGNORE;IGNORE;IGNORE
<D2>     IGNORE;IGNORE;IGNORE
<D3>     IGNORE;IGNORE;IGNORE
<D4>     IGNORE;IGNORE;IGNORE
<NK>     IGNORE;IGNORE;IGNORE
<SY>     IGNORE;IGNORE;IGNORE
<EB>     IGNORE;IGNORE;IGNORE
<CN>     IGNORE;IGNORE;IGNORE
<EM>     IGNORE;IGNORE;IGNORE
<SB>     IGNORE;IGNORE;IGNORE
<EC>     IGNORE;IGNORE;IGNORE
<FS>     IGNORE;IGNORE;IGNORE
<GS>     IGNORE;IGNORE;IGNORE
<RS>     IGNORE;IGNORE;IGNORE
<US>     IGNORE;IGNORE;IGNORE
<DT>     IGNORE;IGNORE;IGNORE
<PA>     IGNORE;IGNORE;IGNORE
<HO>     IGNORE;IGNORE;IGNORE
<BH>     IGNORE;IGNORE;IGNORE
<NH>     IGNORE;IGNORE;IGNORE
<IN>     IGNORE;IGNORE;IGNORE
<NL>     IGNORE;IGNORE;IGNORE
```

```
<SA>    IGNORE;IGNORE;IGNORE
<ES>    IGNORE;IGNORE;IGNORE
<HS>    IGNORE;IGNORE;IGNORE
<HJ>    IGNORE;IGNORE;IGNORE
<VS>    IGNORE;IGNORE;IGNORE
<PD>    IGNORE;IGNORE;IGNORE
<PU>    IGNORE;IGNORE;IGNORE
<RI>    IGNORE;IGNORE;IGNORE
<S2>    IGNORE;IGNORE;IGNORE
<S3>    IGNORE;IGNORE;IGNORE
<DC>    IGNORE;IGNORE;IGNORE
<P1>    IGNORE;IGNORE;IGNORE
<P2>    IGNORE;IGNORE;IGNORE
<TS>    IGNORE;IGNORE;IGNORE
<CC>    IGNORE;IGNORE;IGNORE
<MW>    IGNORE;IGNORE;IGNORE
<SG>    IGNORE;IGNORE;IGNORE
<EG>    IGNORE;IGNORE;IGNORE
<SS>    IGNORE;IGNORE;IGNORE
<GC>    IGNORE;IGNORE;IGNORE
<SC>    IGNORE;IGNORE;IGNORE
<CI>    IGNORE;IGNORE;IGNORE
<ST>    IGNORE;IGNORE;IGNORE
<OC>    IGNORE;IGNORE;IGNORE
<PM>    IGNORE;IGNORE;IGNORE
<AC>    IGNORE;IGNORE;IGNORE
<__>    IGNORE;IGNORE;IGNORE                                        1
<"!>    IGNORE;IGNORE;IGNORE
<"'>    IGNORE;IGNORE;IGNORE
<"/>>   IGNORE;IGNORE;IGNORE
<"?>    IGNORE;IGNORE;IGNORE
<"->    IGNORE;IGNORE;IGNORE
<"(>    IGNORE;IGNORE;IGNORE
<".>    IGNORE;IGNORE;IGNORE
<":>    IGNORE;IGNORE;IGNORE
<"//>   IGNORE;IGNORE;IGNORE
<"0>    IGNORE;IGNORE;IGNORE
<",>    IGNORE;IGNORE;IGNORE
<"_>    IGNORE;IGNORE;IGNORE
<"">    IGNORE;IGNORE;IGNORE
<"<>    IGNORE;IGNORE;IGNORE
<";>    IGNORE;IGNORE;IGNORE
<"=>    IGNORE;IGNORE;IGNORE
<"1>    IGNORE;IGNORE;IGNORE
<"2>    IGNORE;IGNORE;IGNORE
<Fd>    IGNORE;IGNORE;IGNORE
<Bd>    IGNORE;IGNORE;IGNORE
<Fl>    IGNORE;IGNORE;IGNORE
<Li>    IGNORE;IGNORE;IGNORE
<//f>   IGNORE;IGNORE;IGNORE
<0s>    IGNORE;IGNORE;IGNORE
<1s>    IGNORE;IGNORE;IGNORE
<2s>    IGNORE;IGNORE;IGNORE
<3s>    IGNORE;IGNORE;IGNORE
<4s>    IGNORE;IGNORE;IGNORE
<5s>    IGNORE;IGNORE;IGNORE
<6s>    IGNORE;IGNORE;IGNORE
```

```
<7s>     IGNORE;IGNORE;IGNORE
<8s>     IGNORE;IGNORE;IGNORE
<9s>     IGNORE;IGNORE;IGNORE
<0S>     IGNORE;IGNORE;IGNORE
<4S>     IGNORE;IGNORE;IGNORE
<5S>     IGNORE;IGNORE;IGNORE
<6S>     IGNORE;IGNORE;IGNORE
<7S>     IGNORE;IGNORE;IGNORE
<8S>     IGNORE;IGNORE;IGNORE
<9S>     IGNORE;IGNORE;IGNORE
<+S>     IGNORE;IGNORE;IGNORE
<-S>     IGNORE;IGNORE;IGNORE
<1h>     IGNORE;IGNORE;IGNORE
<2h>     IGNORE;IGNORE;IGNORE
<3h>     IGNORE;IGNORE;IGNORE
<4h>     IGNORE;IGNORE;IGNORE
<1j>     IGNORE;IGNORE;IGNORE
<2j>     IGNORE;IGNORE;IGNORE
<3j>     IGNORE;IGNORE;IGNORE
<4j>     IGNORE;IGNORE;IGNORE
<UA>     IGNORE;IGNORE;IGNORE
<UB>     IGNORE;IGNORE;IGNORE
<yr>     IGNORE;IGNORE;IGNORE
<.6>     IGNORE;IGNORE;IGNORE
<<6>     IGNORE;IGNORE;IGNORE
</>6>    IGNORE;IGNORE;IGNORE
<,6>     IGNORE;IGNORE;IGNORE
<&6>     IGNORE;IGNORE;IGNORE
<(S>     IGNORE;IGNORE;IGNORE
<)S>     IGNORE;IGNORE;IGNORE
<UNDEFINED>      IGNORE;IGNORE;IGNORE
<0>
<1>      <1>
<1S>     <1>
<2>      <2>
<2S>     <2>
<3>      <3>
<3S>     <3>
<4>
<5>
<6>
<7>
<8>
<9>
<A>      <A>;<NO-ACCENT>;<CAPITAL>
<a>      <A>;<NO-ACCENT>;<SMALL>
<A'>     <A>;<ACUTE>;<CAPITAL>
<a'>     <A>;<ACUTE>;<SMALL>
<A!>     <A>;<GRAVE>;<CAPITAL>
<a!>     <A>;<GRAVE>;<SMALL>
<A/>>    <A>;<CIRCUMFLEX>;<CAPITAL>
<a/>>    <A>;<CIRCUMFLEX>;<SMALL>
<A?>     <A>;<TILDE>;<CAPITAL>
<a?>     <A>;<TILDE>;<SMALL>
<A->     <A>;<MACRON>;<CAPITAL>
<a->     <A>;<MACRON>;<SMALL>
<A(>     <A>;<BREVE>;<CAPITAL>
```

```
<a(>    <A>;<BREVE>;<SMALL>
<A_>    <A>;<UNDERLINE>;<CAPITAL>
<a_>    <A>;<UNDERLINE>;<SMALL>
<A;>    <A>;<OGONEK>;<CAPITAL>
<a;>    <A>;<OGONEK>;<SMALL>
<A<>    <A>;<CARON>;<CAPITAL>
<a<>    <A>;<CARON>;<SMALL>
<’A>    <A>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<’a>    <A>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<A1>    <A>;<ACC1>;<CAPITAL>
<a1>    <A>;<ACC1>;<SMALL>
<A2>    <A>;<ACC2>;<CAPITAL>
<a2>    <A>;<ACC2>;<SMALL>
<B>     <B>;<NO-ACCENT>;<CAPITAL>
<b>     <B>;<NO-ACCENT>;<SMALL>
<B.>    <B>;<DOT>;<CAPITAL>
<b.>    <B>;<DOT>;<SMALL>
<B_>    <B>;<UNDERLINE>;<CAPITAL>
<b_>    <B>;<UNDERLINE>;<SMALL>
<’B>    <B>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<’b>    <B>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<C>     <C>;<NO-ACCENT>;<CAPITAL>
<c>     <C>;<NO-ACCENT>;<SMALL>
<C’>    <C>;<ACUTE>;<CAPITAL>
<c’>    <C>;<ACUTE>;<SMALL>
<C/>>   <C>;<CIRCUMFLEX>;<CAPITAL>
<c/>>   <C>;<CIRCUMFLEX>;<SMALL>
<C.>    <C>;<DOT>;<CAPITAL>
<c.>    <C>;<DOT>;<SMALL>
<C,>    <C>;<CEDILLA>;<CAPITAL>
<c,>    <C>;<CEDILLA>;<SMALL>
<C<>    <C>;<CARON>;<CAPITAL>
<c<>    <C>;<CARON>;<SMALL>
<D>     <D>;<NO-ACCENT>;<CAPITAL>
<d>     <D>;<NO-ACCENT>;<SMALL>
<D.>    <D>;<DOT>;<CAPITAL>
<d.>    <D>;<DOT>;<SMALL>
<D_>    <D>;<UNDERLINE>;<CAPITAL>
<d_>    <D>;<UNDERLINE>;<SMALL>
<D//>   <D>;<STROKE>;<CAPITAL>
<d//>   <D>;<STROKE>;<SMALL>
<D;>    <D>;<OGONEK>;<CAPITAL>
<d;>    <D>;<OGONEK>;<SMALL>
<D<>    <D>;<CARON>;<CAPITAL>
<d<>    <D>;<CARON>;<SMALL>
<’D>    <D>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<’d>    <D>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<D->    <D>;<SPECIAL>;<CAPITAL>
<d->    <D>;<SPECIAL>;<SMALL>
<E>     <E>;<NO-ACCENT>;<CAPITAL>
<e>     <E>;<NO-ACCENT>;<SMALL>
<E’>    <E>;<ACUTE>;<CAPITAL>
<e’>    <E>;<ACUTE>;<SMALL>
<E!>    <E>;<GRAVE>;<CAPITAL>
<e!>    <E>;<GRAVE>;<SMALL>
<E/>>   <E>;<CIRCUMFLEX>;<CAPITAL>
<e/>>   <E>;<CIRCUMFLEX>;<SMALL>
```

```
<E?>    <E>;<TILDE>;<CAPITAL>
<e?>    <E>;<TILDE>;<SMALL>
<E->    <E>;<MACRON>;<CAPITAL>
<e->    <E>;<MACRON>;<SMALL>
<E(>    <E>;<BREVE>;<CAPITAL>
<e(>    <E>;<BREVE>;<SMALL>
<E.>    <E>;<DOT>;<CAPITAL>
<e.>    <E>;<DOT>;<SMALL>
<E:>    <E>;<DIAERESIS>;<CAPITAL>
<e:>    <E>;<DIAERESIS>;<SMALL>
<E_>    <E>;<UNDERLINE>;<CAPITAL>
<e_>    <E>;<UNDERLINE>;<SMALL>
<E;>    <E>;<OGONEK>;<CAPITAL>
<e;>    <E>;<OGONEK>;<SMALL>
<E<>    <E>;<CARON>;<CAPITAL>
<e<>    <E>;<CARON>;<SMALL>
<F>     <F>;<NO-ACCENT>;<CAPITAL>
<f>     <F>;<NO-ACCENT>;<SMALL>
<F.>    <F>;<DOT>;<CAPITAL>
<f.>    <F>;<DOT>;<SMALL>
<ff>    <FF+>;<SPECIAL>;<SMALL>
<fi>    <FI+>;<SPECIAL>;<SMALL>
<fl>    <FL+>;<SPECIAL>;<SMALL>
<ft>    <FT+>;<SPECIAL>;<SMALL>
<G>     <G>;<NO-ACCENT>;<CAPITAL>
<g>     <G>;<NO-ACCENT>;<SMALL>
<G'>    <G>;<ACUTE>;<CAPITAL>
<g'>    <G>;<ACUTE>;<SMALL>
<G/>>   <G>;<CIRCUMFLEX>;<CAPITAL>
<g/>>   <G>;<CIRCUMFLEX>;<SMALL>
<G->    <G>;<MACRON>;<CAPITAL>
<g->    <G>;<MACRON>;<SMALL>
<G(>    <G>;<BREVE>;<CAPITAL>
<g(>    <G>;<BREVE>;<SMALL>
<G.>    <G>;<DOT>;<CAPITAL>
<g.>    <G>;<DOT>;<SMALL>
<G,>    <G>;<CEDILLA>;<CAPITAL>
<g,>    <G>;<CEDILLA>;<SMALL>
<G//>   <G>;<STROKE>;<CAPITAL>
<g//>   <G>;<STROKE>;<SMALL>
<G<>    <G>;<CARON>;<CAPITAL>
<g<>    <G>;<CARON>;<SMALL>
<'G>    <G>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<'g>    <G>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<H>     <H>;<NO-ACCENT>;<CAPITAL>
<h>     <H>;<NO-ACCENT>;<SMALL>
<H/>>   <H>;<CIRCUMFLEX>;<CAPITAL>
<h/>>   <H>;<CIRCUMFLEX>;<SMALL>
<H.>    <H>;<DOT>;<CAPITAL>
<h.>    <H>;<DOT>;<SMALL>
<H:>    <H>;<DIAERESIS>;<CAPITAL>
<h:>    <H>;<DIAERESIS>;<SMALL>
<H,>    <H>;<CEDILLA>;<CAPITAL>
<h,>    <H>;<CEDILLA>;<SMALL>
<H//>   <H>;<STROKE>;<CAPITAL>
<h//>   <H>;<STROKE>;<SMALL>
<H;>    <H>;<OGONEK>;<CAPITAL>
```

```
<h;>    <H>;<OGONEK>;<SMALL>
<I>     <I>;<NO-ACCENT>;<CAPITAL>
<i>     <I>;<NO-ACCENT>;<SMALL>
<I'>    <I>;<ACUTE>;<CAPITAL>
<i'>    <I>;<ACUTE>;<SMALL>
<I!>    <I>;<GRAVE>;<CAPITAL>
<i!>    <I>;<GRAVE>;<SMALL>
<I/>>   <I>;<CIRCUMFLEX>;<CAPITAL>
<i/>>   <I>;<CIRCUMFLEX>;<SMALL>
<I?>    <I>;<TILDE>;<CAPITAL>
<i?>    <I>;<TILDE>;<SMALL>
<I->    <I>;<MACRON>;<CAPITAL>
<i->    <I>;<MACRON>;<SMALL>
<I(>    <I>;<BREVE>;<CAPITAL>
<i(>    <I>;<BREVE>;<SMALL>
<I.>    <I>;<DOT>;<CAPITAL>
<i.>    <I>;<DOT>;<SMALL>
<I:>    <I>;<DIAERESIS>;<CAPITAL>
<i:>    <I>;<DIAERESIS>;<SMALL>
<I;>    <I>;<OGONEK>;<CAPITAL>
<i;>    <I>;<OGONEK>;<SMALL>
<I<>    <I>;<CARON>;<CAPITAL>
<i<>    <I>;<CARON>;<SMALL>
<I-J>   <I><J>;<I-J><I-J>;<CAPITAL><CAPITAL>
<i-j>   <I><J>;<I-J><I-J>;<SMALL><SMALL>
<IJ>    <I><J>;<IJ><IJ>;<CAPITAL><CAPITAL>
<ij>    <I><J>;<IJ><IJ>;<SMALL><SMALL>
<J>     <J>;<NO-ACCENT>;<CAPITAL>
<j>     <J>;<NO-ACCENT>;<SMALL>
<J/>>   <J>;<CIRCUMFLEX>;<CAPITAL>
<j/>>   <J>;<CIRCUMFLEX>;<SMALL>
<J(>    <J>;<BREVE>;<CAPITAL>
<j(>    <J>;<BREVE>;<SMALL>
<'J>    <J>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<'j>    <J>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<K>     <K>;<NO-ACCENT>;<CAPITAL>
<k>     <K>;<NO-ACCENT>;<SMALL>
<K'>    <K>;<ACUTE>;<CAPITAL>
<k'>    <K>;<ACUTE>;<SMALL>
<K.>    <K>;<DOT>;<CAPITAL>
<k.>    <K>;<DOT>;<SMALL>
<K,>    <K>;<CEDILLA>;<CAPITAL>
<k,>    <K>;<CEDILLA>;<SMALL>
<K_>    <K>;<UNDERLINE>;<CAPITAL>
<k_>    <K>;<UNDERLINE>;<SMALL>
<K;>    <K>;<OGONEK>;<CAPITAL>
<k;>    <K>;<OGONEK>;<SMALL>
<K<>    <K>;<CARON>;<CAPITAL>
<k<>    <K>;<CARON>;<SMALL>
<L>     <L>;<NO-ACCENT>;<CAPITAL>
<l>     <L>;<NO-ACCENT>;<SMALL>
<L'>    <L>;<ACUTE>;<CAPITAL>
<l'>    <L>;<ACUTE>;<SMALL>
<L.>    <L>;<DOT>;<CAPITAL>
<l.>    <L>;<DOT>;<SMALL>
<L,>    <L>;<CEDILLA>;<CAPITAL>
<l,>    <L>;<CEDILLA>;<SMALL>
```

```
<L_>     <L>;<UNDERLINE>;<CAPITAL>
<l_>     <L>;<UNDERLINE>;<SMALL>
<L//>    <L>;<STROKE>;<CAPITAL>
<l//>    <L>;<STROKE>;<SMALL>
<L<>     <L>;<CARON>;<CAPITAL>
<l<>     <L>;<CARON>;<SMALL>
<M>      <M>;<NO-ACCENT>;<CAPITAL>
<m>      <M>;<NO-ACCENT>;<SMALL>
<M’>     <M>;<ACUTE>;<CAPITAL>
<m’>     <M>;<ACUTE>;<SMALL>
<M.>     <M>;<DOT>;<CAPITAL>
<m.>     <M>;<DOT>;<SMALL>
<N>      <N>;<NO-ACCENT>;<CAPITAL>
<n>      <N>;<NO-ACCENT>;<SMALL>
<N’>     <N>;<ACUTE>;<CAPITAL>
<n’>     <N>;<ACUTE>;<SMALL>
<N?>     <N>;<TILDE>;<CAPITAL>
<n?>     <N>;<TILDE>;<SMALL>
<N.>     <N>;<DOT>;<CAPITAL>
<n.>     <N>;<DOT>;<SMALL>
<N,>     <N>;<CEDILLA>;<CAPITAL>
<n,>     <N>;<CEDILLA>;<SMALL>
<N_>     <N>;<UNDERLINE>;<CAPITAL>
<n_>     <N>;<UNDERLINE>;<SMALL>
<N<>     <N>;<CARON>;<CAPITAL>
<n<>     <N>;<CARON>;<SMALL>
<’n>     <N>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
<N-G>    <N><G>;<N-G><N-G>;<CAPITAL><CAPITAL>
<n-g>    <N><G>;<N-G><N-G>;<SMALL><SMALL>
<NG>     <N><G>;<NG><NG>;<CAPITAL><CAPITAL>
<ng>     <N><G>;<NG><NG>;<SMALL><SMALL>
<O>      <O>;<NO-ACCENT>;<CAPITAL>
<o>      <O>;<NO-ACCENT>;<SMALL>
<O’>     <O>;<ACUTE>;<CAPITAL>
<o’>     <O>;<ACUTE>;<SMALL>
<O!>     <O>;<GRAVE>;<CAPITAL>
<o!>     <O>;<GRAVE>;<SMALL>
<O/>>    <O>;<CIRCUMFLEX>;<CAPITAL>
<o/>>    <O>;<CIRCUMFLEX>;<SMALL>
<O?>     <O>;<TILDE>;<CAPITAL>
<o?>     <O>;<TILDE>;<SMALL>
<O->     <O>;<MACRON>;<CAPITAL>
<o->     <O>;<MACRON>;<SMALL>
<O(>     <O>;<BREVE>;<CAPITAL>
<o(>     <O>;<BREVE>;<SMALL>
<O_>     <O>;<UNDERLINE>;<CAPITAL>
<o_>     <O>;<UNDERLINE>;<SMALL>
<O;>     <O>;<OGONEK>;<CAPITAL>
<o;>     <O>;<OGONEK>;<SMALL>
<O<>     <O>;<CARON>;<CAPITAL>
<o<>     <O>;<CARON>;<SMALL>
<O1>     <O>;<ACC1>;<CAPITAL>
<o1>     <O>;<ACC1>;<SMALL>
<O-E>    <O><E>;<O-E><O-E>;<CAPITAL><CAPITAL>
<o-e>    <O><E>;<O-E><O-E>;<SMALL><SMALL>
<OE>     <O><E>;<OE><OE>;<CAPITAL><CAPITAL>
<oe>     <O><E>;<OE><OE>;<SMALL><SMALL>
```

```
<P>      <P>;<NO-ACCENT>;<CAPITAL>
<p>      <P>;<NO-ACCENT>;<SMALL>
<P'>     <P>;<ACUTE>;<CAPITAL>
<p'>     <P>;<ACUTE>;<SMALL>
<Q>      <Q>;<NO-ACCENT>;<CAPITAL>
<q>      <Q>;<NO-ACCENT>;<SMALL>
<kk>     <Q>;<SPECIAL>;<SMALL>
<R>      <R>;<NO-ACCENT>;<CAPITAL>
<r>      <R>;<NO-ACCENT>;<SMALL>
<R'>     <R>;<ACUTE>;<CAPITAL>
<r'>     <R>;<ACUTE>;<SMALL>
<R.>     <R>;<DOT>;<CAPITAL>
<r.>     <R>;<DOT>;<SMALL>
<R,>     <R>;<CEDILLA>;<CAPITAL>
<r,>     <R>;<CEDILLA>;<SMALL>
<R_>     <R>;<UNDERLINE>;<CAPITAL>
<r_>     <R>;<UNDERLINE>;<SMALL>
<R<>     <R>;<CARON>;<CAPITAL>
<r<>     <R>;<CARON>;<SMALL>
<S>      <S>;<NO-ACCENT>;<CAPITAL>
<s>      <S>;<NO-ACCENT>;<SMALL>
<S'>     <S>;<ACUTE>;<CAPITAL>
<s'>     <S>;<ACUTE>;<SMALL>
<S/>>    <S>;<CIRCUMFLEX>;<CAPITAL>
<s/>>    <S>;<CIRCUMFLEX>;<SMALL>
<S.>     <S>;<DOT>;<CAPITAL>
<s.>     <S>;<DOT>;<SMALL>
<S,>     <S>;<CEDILLA>;<CAPITAL>
<s,>     <S>;<CEDILLA>;<SMALL>
<S;>     <S>;<OGONEK>;<CAPITAL>
<s;>     <S>;<OGONEK>;<SMALL>
<S<>     <S>;<CARON>;<CAPITAL>
<s<>     <S>;<CARON>;<SMALL>
<ss>     <S><S>;<ss><ss>;<SMALL><SMALL>                1
<s-s>    <S><S>;<s-s><s-s>;<SMALL><SMALL>              1
<st>     <ST+>;<SPECIAL>;<SMALL>
<T>      <T>;<NO-ACCENT>;<CAPITAL>
<t>      <T>;<NO-ACCENT>;<SMALL>
<T.>     <T>;<DOT>;<CAPITAL>
<t.>     <T>;<DOT>;<SMALL>
<T,>     <T>;<CEDILLA>;<CAPITAL>
<t,>     <T>;<CEDILLA>;<SMALL>
<T_>     <T>;<UNDERLINE>;<CAPITAL>
<t_>     <T>;<UNDERLINE>;<SMALL>
<T//>    <T>;<STROKE>;<CAPITAL>
<t//>    <T>;<STROKE>;<SMALL>
<T<>     <T>;<CARON>;<CAPITAL>
<t<>     <T>;<CARON>;<SMALL>
<T-H>    <T><H>;<T-H><T-H>;<CAPITAL><CAPITAL>
<t-h>    <T><H>;<T-H><T-H>;<SMALL><SMALL>
<TH>     <T><H>;<TH><TH>;<CAPITAL><CAPITAL>
<th>     <T><H>;<TH><TH>;<SMALL><SMALL>
<U>      <U>;<NO-ACCENT>;<CAPITAL>
<u>      <U>;<NO-ACCENT>;<SMALL>
<U'>     <U>;<ACUTE>;<CAPITAL>
<u'>     <U>;<ACUTE>;<SMALL>
<U!>     <U>;<GRAVE>;<CAPITAL>
```

```
<u!>     <U>;<GRAVE>;<SMALL>
<U/>>    <U>;<CIRCUMFLEX>;<CAPITAL>
<u/>>    <U>;<CIRCUMFLEX>;<SMALL>
<U?>     <U>;<TILDE>;<CAPITAL>
<u?>     <U>;<TILDE>;<SMALL>
<U->     <U>;<MACRON>;<CAPITAL>
<u->     <U>;<MACRON>;<SMALL>
<U(>     <U>;<BREVE>;<CAPITAL>
<u(>     <U>;<BREVE>;<SMALL>
<U;>     <U>;<OGONEK>;<CAPITAL>                                  1
<u;>     <U>;<OGONEK>;<SMALL>
<U<>     <U>;<CARON>;<CAPITAL>
<u<>     <U>;<CARON>;<SMALL>
<U0>     <U>;<RING>;<CAPITAL>                                    1
<u0>     <U>;<RING>;<SMALL>                                      1
<V>      <V>;<NO-ACCENT>;<CAPITAL>
<v>      <V>;<NO-ACCENT>;<SMALL>
<V?>     <V>;<TILDE>;<CAPITAL>
<v?>     <V>;<TILDE>;<SMALL>
<W>      <W>;<NO-ACCENT>;<CAPITAL>
<w>      <W>;<NO-ACCENT>;<SMALL>
<W'>     <W>;<ACUTE>;<CAPITAL>
<w'>     <W>;<ACUTE>;<SMALL>
<W/>>    <W>;<CIRCUMFLEX>;<CAPITAL>
<w/>>    <W>;<CIRCUMFLEX>;<SMALL>
<W.>     <W>;<DOT>;<CAPITAL>
<w.>     <W>;<DOT>;<SMALL>
<W:>     <W>;<DIAERESIS>;<CAPITAL>
<w:>     <W>;<DIAERESIS>;<SMALL>
<X>      <X>;<NO-ACCENT>;<CAPITAL>
<x>      <X>;<NO-ACCENT>;<SMALL>
<X.>     <X>;<DOT>;<CAPITAL>
<x.>     <X>;<DOT>;<SMALL>
<X:>     <X>;<DIAERESIS>;<CAPITAL>
<x:>     <X>;<DIAERESIS>;<SMALL>
<Y>      <Y>;<NO-ACCENT>;<CAPITAL>
<y>      <Y>;<NO-ACCENT>;<SMALL>
<Y'>     <Y>;<ACUTE>;<CAPITAL>
<y'>     <Y>;<ACUTE>;<SMALL>
<Y!>     <Y>;<GRAVE>;<CAPITAL>
<y!>     <Y>;<GRAVE>;<SMALL>
<Y/>>    <Y>;<CIRCUMFLEX>;<CAPITAL>
<y/>>    <Y>;<CIRCUMFLEX>;<SMALL>
<Y.>     <Y>;<DOT>;<CAPITAL>
<y.>     <Y>;<DOT>;<SMALL>
<'Y>     <Y>;<PRECEDED-BY-APOSTROPHE>;<CAPITAL>
<'y>     <Y>;<PRECEDED-BY-APOSTROPHE>;<SMALL>
% <U:> and <U"> are treated as <Y> in Danish                     1
<U:>     <Y>;<ACC11>;<CAPITAL>
<u:>     <Y>;<ACC11>;<SMALL>
<U">     <Y>;<ACC12>;<CAPITAL>
<u">     <Y>;<ACC12>;<SMALL>
<Z>      <Z>;<NO-ACCENT>;<CAPITAL>
<z>      <Z>;<NO-ACCENT>;<SMALL>
<Z'>     <Z>;<ACUTE>;<CAPITAL>
<z'>     <Z>;<ACUTE>;<SMALL>
<Z/>>    <Z>;<CIRCUMFLEX>;<CAPITAL>
```

```
<z/>>   <Z>;<CIRCUMFLEX>;<SMALL>
<Z(>    <Z>;<BREVE>;<CAPITAL>
<z(>    <Z>;<BREVE>;<SMALL>
<Z.>    <Z>;<DOT>;<CAPITAL>
<z.>    <Z>;<DOT>;<SMALL>
<Z_>    <Z>;<UNDERLINE>;<CAPITAL>
<z_>    <Z>;<UNDERLINE>;<SMALL>
<Z//>   <Z>;<STROKE>;<CAPITAL>
<z//>   <Z>;<STROKE>;<SMALL>
<Z<>    <Z>;<CARON>;<CAPITAL>
<z<>    <Z>;<CARON>;<SMALL>
% <AE> is treated as a separate letter in Danish                1
<AE>    <AE>;<NO-ACCENT>;<CAPITAL>
<ae>    <AE>;<NO-ACCENT>;<SMALL>
<A:>    <AE>;<DIAERESIS>;<CAPITAL>
<a:>    <AE>;<DIAERESIS>;<SMALL>
<A3>    <AE>;<ACC3>;<CAPITAL>
<a3>    <AE>;<ACC3>;<SMALL>
% <O//> is treated as a separate letter in Danish               1
<O//>   <O//>;<NO-ACCENT>;<CAPITAL>
<o//>   <O//>;<NO-ACCENT>;<SMALL>
<O:>    <O//>;<DIAERESIS>;<CAPITAL>
<o:>    <O//>;<DIAERESIS>;<SMALL>
<O">    <O//>;<DOUBLE-ACUTE>;<CAPITAL>
<o">    <O//>;<DOUBLE-ACUTE>;<SMALL>
% <AA> is treated as a separate letter in Danish                1
<AA>    <AA>;<NO-ACCENT>;<CAPITAL>
<aa>    <AA>;<NO-ACCENT>;<SMALL>
<A-A>   <AA>;<ACC1>;<CAPITAL>
<A-a>   <AA>;<ACC1>;<BOTH>
<a-a>   <AA>;<ACC1>;<SMALL>

<A=>    <A=>;<CYRILLIC>;<CAPITAL>
<a=>    <A=>;<CYRILLIC>;<SMALL>
<B=>    <B=>;<CYRILLIC>;<CAPITAL>
<b=>    <B=>;<CYRILLIC>;<SMALL>
<V=>    <V=>;<CYRILLIC>;<CAPITAL>
<v=>    <V=>;<CYRILLIC>;<SMALL>
<G=>    <G=>;<CYRILLIC>;<CAPITAL>
<g=>    <G=>;<CYRILLIC>;<SMALL>
<G%>    <G=>;<ALPHA-1>;<CAPITAL>
<g%>    <G=>;<ALPHA-1>;<SMALL>
<D=>    <D=>;<CYRILLIC>;<CAPITAL>
<d=>    <D=>;<CYRILLIC>;<SMALL>
<D%>    <D%>;<ALPHA-1>;<CAPITAL>
<d%>    <D%>;<ALPHA-1>;<SMALL>
<E=>    <E=>;<CYRILLIC>;<CAPITAL>
<e=>    <E=>;<CYRILLIC>;<SMALL>
<IO>    <E=>;<SPECIAL>;<CAPITAL>
<io>    <E=>;<SPECIAL>;<SMALL>
<IE>    <IE>;<SPECIAL>;<CAPITAL>
<ie>    <IE>;<SPECIAL>;<SMALL>
<Z%>    <Z%>;<ALPHA-1>;<CAPITAL>
<z%>    <Z%>;<ALPHA-1>;<SMALL>
<Z=>    <Z=>;<CYRILLIC>;<CAPITAL>
<z=>    <Z=>;<CYRILLIC>;<SMALL>
<DS>    <DS>;<SPECIAL>;<CAPITAL>                                1
```

```
<ds>     <DS>;<SPECIAL>;<SMALL>                                    1
<I=>     <I=>;<CYRILLIC>;<CAPITAL>
<i=>     <I=>;<CYRILLIC>;<SMALL>
<II>     <II>;<SPECIAL>;<CAPITAL>
<ii>     <II>;<SPECIAL>;<SMALL>
<YI>     <II>;<ALPHA-1>;<CAPITAL>
<yi>     <II>;<ALPHA-1>;<SMALL>
<J%>     <J%>;<ALPHA-1>;<CAPITAL>
<j%>     <J%>;<ALPHA-1>;<SMALL>
<J=>     <J=>;<CYRILLIC>;<CAPITAL>
<j=>     <J=>;<CYRILLIC>;<SMALL>
<K=>     <K=>;<CYRILLIC>;<CAPITAL>
<k=>     <K=>;<CYRILLIC>;<SMALL>
<KJ>     <K=>;<SPECIAL>;<CAPITAL>
<kj>     <K=>;<SPECIAL>;<SMALL>
<L=>     <L=>;<CYRILLIC>;<CAPITAL>
<l=>     <L=>;<CYRILLIC>;<SMALL>
<LJ>     <LJ>;<SPECIAL>;<CAPITAL>
<lj>     <LJ>;<SPECIAL>;<SMALL>
<M=>     <M=>;<CYRILLIC>;<CAPITAL>
<m=>     <M=>;<CYRILLIC>;<SMALL>
<N=>     <N=>;<CYRILLIC>;<CAPITAL>
<n=>     <N=>;<CYRILLIC>;<SMALL>
<NJ>     <NJ>;<SPECIAL>;<CAPITAL>
<nj>     <NJ>;<SPECIAL>;<SMALL>
<O=>     <O=>;<CYRILLIC>;<CAPITAL>
<o=>     <O=>;<CYRILLIC>;<SMALL>
<P=>     <P=>;<CYRILLIC>;<CAPITAL>
<p=>     <P=>;<CYRILLIC>;<SMALL>
<R=>     <R=>;<CYRILLIC>;<CAPITAL>
<r=>     <R=>;<CYRILLIC>;<SMALL>
<S=>     <S=>;<CYRILLIC>;<CAPITAL>
<s=>     <S=>;<CYRILLIC>;<SMALL>
<T=>     <T=>;<CYRILLIC>;<CAPITAL>
<t=>     <T=>;<CYRILLIC>;<SMALL>
<Ts>     <Ts>;<SPECIAL>;<CAPITAL>
<ts>     <Ts>;<SPECIAL>;<SMALL>
<U=>     <U=>;<CYRILLIC>;<CAPITAL>
<u=>     <U=>;<CYRILLIC>;<SMALL>
<V%>     <V%>;<ALPHA-1>;<CAPITAL>
<v%>     <V%>;<ALPHA-1>;<SMALL>
<F=>     <F=>;<CYRILLIC>;<CAPITAL>
<f=>     <F=>;<CYRILLIC>;<SMALL>
<H=>     <H=>;<CYRILLIC>;<CAPITAL>
<h=>     <H=>;<CYRILLIC>;<SMALL>
<C=>     <C=>;<CYRILLIC>;<CAPITAL>
<c=>     <C=>;<CYRILLIC>;<SMALL>
<C%>     <C%>;<ALPHA-1>;<CAPITAL>
<c%>     <C%>;<ALPHA-1>;<SMALL>
<DZ>     <DZ>;<SPECIAL>;<CAPITAL>
<dz>     <DZ>;<SPECIAL>;<SMALL>
<S%>     <S%>;<ALPHA-1>;<CAPITAL>
<s%>     <S%>;<ALPHA-1>;<SMALL>
<Sc>     <Sc>;<SPECIAL>;<CAPITAL>
<sc>     <Sc>;<SPECIAL>;<SMALL>
<=’>     <=’>;<ACUTE>;<SMALL>
<=">     <=’>;<DOUBLE-ACUTE>;<CAPITAL>
```

```
<Y=>    <Y=>;<CYRILLIC>;<CAPITAL>
<y=>    <Y=>;<CYRILLIC>;<SMALL>
<%'>    <%'>;<ACUTE>;<SMALL>
<%">    <%'>;<DOUBLE-ACUTE>;<CAPITAL>
<JE>    <JE>;<SPECIAL>;<CAPITAL>
<je>    <JE>;<SPECIAL>;<SMALL>
<JU>    <JU>;<SPECIAL>;<CAPITAL>
<ju>    <JU>;<SPECIAL>;<SMALL>
<JA>    <JA>;<SPECIAL>;<CAPITAL>
<ja>    <JA>;<SPECIAL>;<SMALL>
<A*>    <A*>;<GREEK>;<CAPITAL>                1
<a*>    <A*>;<GREEK>;<SMALL>
<A%>    <A*>;<ALPHA-1>;<CAPITAL>
<a%>    <A*>;<ALPHA-1>;<SMALL>
<B*>    <B*>;<GREEK>;<CAPITAL>
<b*>    <B*>;<GREEK>;<SMALL>
<G*>    <G*>;<GREEK>;<CAPITAL>
<g*>    <G*>;<GREEK>;<SMALL>
<D*>    <D*>;<GREEK>;<CAPITAL>
<d*>    <D*>;<GREEK>;<SMALL>
<E*>    <E*>;<GREEK>;<CAPITAL>
<e*>    <E*>;<GREEK>;<SMALL>
<E%>    <E*>;<ALPHA-1>;<CAPITAL>
<e%>    <E*>;<ALPHA-1>;<SMALL>
<Z*>    <Z*>;<GREEK>;<CAPITAL>
<z*>    <Z*>;<GREEK>;<SMALL>
<Y*>    <Y*>;<GREEK>;<CAPITAL>
<y*>    <Y*>;<GREEK>;<SMALL>
<Y%>    <Y*>;<ALPHA-1>;<CAPITAL>
<y%>    <Y*>;<ALPHA-1>;<SMALL>
<H*>    <H*>;<GREEK>;<CAPITAL>
<h*>    <H*>;<GREEK>;<SMALL>
<I*>    <I*>;<GREEK>;<CAPITAL>                1
<J*>    <I*>;<GREEK>;<CAPITAL>                1
<i*>    <I*>;<GREEK>;<SMALL>                  1
<j*>    <I*>;<GREEK>;<SMALL>                  1
<I%>    <I*>;<ALPHA-1>;<CAPITAL>              1
<i%>    <I*>;<ALPHA-1>;<SMALL>                1
<K*>    <K*>;<GREEK>;<CAPITAL>                1
<k*>    <K*>;<GREEK>;<SMALL>                  1
<L*>    <L*>;<GREEK>;<CAPITAL>                1
<l*>    <L*>;<GREEK>;<SMALL>                  1
<M*>    <M*>;<GREEK>;<CAPITAL>                1
<m*>    <M*>;<GREEK>;<SMALL>                  1
<N*>    <N*>;<GREEK>;<CAPITAL>                1
<n*>    <N*>;<GREEK>;<SMALL>                  1
<C*>    <C*>;<GREEK>;<CAPITAL>                1
<c*>    <C*>;<GREEK>;<SMALL>                  1
<O*>    <O*>;<GREEK>;<CAPITAL>
<o*>    <O*>;<GREEK>;<SMALL>
<O%>    <O*>;<ALPHA-1>;<CAPITAL>
<o%>    <O*>;<ALPHA-1>;<SMALL>
<P*>    <P*>;<GREEK>;<CAPITAL>
<p*>    <P*>;<GREEK>;<SMALL>
<R*>    <R*>;<GREEK>;<CAPITAL>
<r*>    <R*>;<GREEK>;<SMALL>
<S*>    <S*>;<GREEK>;<CAPITAL>
```

```
<s*>    <S*>;<GREEK>;<SMALL>
<*s>    <S*>;<SPECIAL>;<SMALL>
<T*>    <T*>;<GREEK>;<CAPITAL>
<t*>    <T*>;<GREEK>;<SMALL>
<U*>    <U*>;<GREEK>;<CAPITAL>
<V*>    <U*>;<GREEK>;<CAPITAL>
<u*>    <U*>;<GREEK>;<SMALL>
<v*>    <U*>;<GREEK>;<SMALL>
<U%>    <U*>;<ALPHA-1>;<CAPITAL>
<u%>    <U*>;<ALPHA-1>;<SMALL>
<F*>    <F*>;<GREEK>;<CAPITAL>
<f*>    <F*>;<GREEK>;<SMALL>
<X*>    <X*>;<GREEK>;<CAPITAL>
<x*>    <X*>;<GREEK>;<SMALL>
<Q*>    <Q*>;<GREEK>;<CAPITAL>
<q*>    <Q*>;<GREEK>;<SMALL>
<W*>    <W*>;<GREEK>;<CAPITAL>
<w*>    <W*>;<GREEK>;<SMALL>
<W%>    <W*>;<ALPHA-1>;<CAPITAL>
<w%>    <W*>;<ALPHA-1>;<SMALL>
<p+>
<v+>
<gf>
<H'>
<aM>
<aH>
<wH>
<ah>
<yH>
<a+>
<b+>
<tm>
<t+>
<tk>
<g+>
<hk>
<x+>
<d+>
<dk>
<r+>
<z+>
<s+>
<sn>
<c+>
<dd>
<tj>
<zH>
<e+>
<i+>
<f+>
<q+>
<k+>
<l+>
<m+>
<n+>
<h+>
<w+>
```

```
<j+>
<y+>
<yf>
<A+>
<B+>
<G+>
<D+>
<H+>
<W+>
<Z+>
<X+>
<Tj>
<J+>
<K%>
<K+>
<L+>
<M%>
<M+>
<N%>
<N+>
<S+>
<E+>
<P%>
<P+>
<Zj>
<ZJ>
<Q+>
<R+>
<Sh>
<T+>
<b4>
<p4>
<m4>
<f4>
<d4>
<t4>
<n4>
<l4>
<g4>
<k4>
<h4>
<j4>
<q4>
<x4>
<zh>
<ch>
<sh>
<r4>
<z4>
<c4>
<s4>
<a4>
<o4>
<e4>
<eh>
<ai>
<ei>
```

```
<au>
<ou>
<an>
<en>
<aN>
<eN>
<er>
<i4>
<u4>
<iu>
<A5>
<a5>
<I5>
<i5>
<U5>
<u5>
<E5>
<e5>
<O5>
<o5>
<ka>
<ga>
<ki>
<gi>
<ku>
<gu>
<ke>
<ge>
<ko>
<go>
<sa>
<za>
<si>
<zi>
<su>
<zu>
<se>
<ze>
<so>
<zo>
<ta>
<da>
<ti>
<di>
<tU>
<tu>
<du>
<te>
<de>
<to>
<do>
<na>
<ni>
<nu>
<ne>
<no>
<ha>
```

```
<ba>
<pa>
<hi>
<bi>
<pi>
<hu>
<bu>
<pu>
<he>
<be>
<pe>
<ho>
<bo>
<po>
<ma>
<mi>
<mu>
<me>
<mo>
<yA>
<ya>
<yU>
<yu>
<yO>
<yo>
<ra>
<ri>
<ru>
<re>
<ro>
<wA>
<wa>
<wi>
<we>
<wo>
<n5>
<a6>
<A6>
<i6>
<I6>
<u6>
<U6>
<e6>
<E6>
<o6>
<O6>
<Ka>
<Ga>
<Ki>
<Gi>
<Ku>
<Gu>
<Ke>
<Ge>
<Ko>
<Go>
<Sa>
```

<Za>
<Si>
<Zi>
<Su>
<Zu>
<Se>
<Ze>
<So>
<Zo>
<Ta>
<Da>
<Ti>
<Di>
<TU>
<Tu>
<Du>
<Te>
<De>
<To>
<Do>
<Na>
<Ni>
<Nu>
<Ne>
<No>
<Ha>
<Ba>
<Pa>
<Hi>
<Bi> 1
<Pi> 1
<Hu>
<Bu>
<Pu>
<He>
<Be>
<Pe>
<Ho>
<Bo>
<Po>
<Ma>
<Mi>
<Mu>
<Me>
<Mo>
<YA>
<Ya>
<YU>
<Yu>
<YO>
<Yo>
<Ra>
<Ri>
<Ru>
<Re>
<Ro>
<WA>

```
<Wa>
<Wi>
<We>
<Wo>
<N6>
<Vu>
<KA>
<KE>

order_end

END LC_COLLATE

LC_MONETARY

% int_curr_symbol according to ISO 4217                          1
int_curr_symbol         "DKK "                                    1
currency_symbol         "kr."                                     1
mon_decimal_point       <,>
mon_thousands_sep       <.>
mon_grouping            3;0
positive_sign           ""
negative_sign           <->
int_frac_digits         2
frac_digits             2
p_cs_precedes           1
p_sep_by_space          1
n_cs_precedes           1
n_sep_by_space          1
p_sign_posn             4
n_sign_posn             4

END LC_MONETARY

LC_NUMERIC

decimal_point           <,>
thousands_sep           <.>
grouping                3;0

END LC_NUMERIC

LC_TIME

abday        "s<o//>n";"man";"tir";"ons";"tor";"fre";"l<o//>r"          1
day          "s<o//>ndag";"mandag";"tirsdag";"onsdag";/                 1
             "torsdag";"fredag";"l<o//>rdag"                            1
abmon        "jan";"feb";"mar";"apr";"maj";"jun";/                      1
             "jul";"aug";"sep";"okt";"nov";"dec"                        1
mon          "januar";"februar";"marts";"april";"maj";"juni";/          1
             "juli";"august";"september";"oktober";"november";"december"  1
d_t_fmt      "%a %d %b %Y %T %Z"                                        1
d_fmt        "%d %b %Y"                                                 1
t_fmt        "%T"                                                       1

% The AM/PM notation is not used in Denmark and thus not allowed.       1
am_pm        "";""
```

```
t_fmt_ampm  ""

END LC_TIME

LC_MESSAGES

% Must be careful to avoid interpreting "nej" (no) as "ja" (yes).

% yesexpr     "^[[:blank:]]*[JjYy][[:alpha:]]*"
% noexpr      "^[[:blank:]]*[Nn][[:alpha:]]*"

yesexpr     "<'//><<(><<(>:blank:<)//><)//>*<<(>JjYy<)//>/
<<(><<(>:alpha:<)//><)//>*"
noexpr      "<'//><<(><<(>:blank:<)//><)//>*<<(>Nn<)//>/
<<(><<(>:alpha:<)//><)//>*"

END LC_MESSAGES
```

F.3.2 fo_DK — (Example) Faroese LC_TIME and LC_MESSAGES

```
escape_char  /
comment_char %
% Danish example national locale for the Faroese language
% Source: Danish Standards Association
% Revision: 1.7 1991-04-26
%
% Only LC_TIME and LC_MESSAGES are specified here, else use the da_DK locale

LC_CTYPE
copy da_DK
END LC_CTYPE

LC_COLLATE
copy da_DK
END LC_COLLATE

LC_MONETARY
copy da_DK
END LC_MONETARY

LC_NUMERIC
copy da_DK
END LC_NUMERIC

LC_TIME

abday       "sun";"m<a'>n";"t<y'>s";"mik";"h<o'>s";"fr<i'>";"ley"
day         "sunnudagur";"m<a'>nadagur";"t<y'>sdagur";/
            "mikudagur";"h<o'>sdagur";"fr<i'>ggjadagur";"leygardagur"
abmon       "jan";"feb";"mar";"apr";"mai";"jun";/
            "jul";"aug";"sep";"okt";"nov";"des"
mon         "januar";"februar";"mars";"apr<i'>l";"mai";"juni";/
            "juli";"august";"september";"oktober";"november";"desember"
d_t_fmt     "%a %d %b %Y %T %Z"
d_fmt       "%d %b %Y"
t_fmt       "%T"
```

```
am_pm        "";""
t_fmt_ampm   ""

END LC_TIME

LC_MESSAGES

% Must be careful to avoid interpreting "nej"/"nei" (no) as "ja" (yes).       1

% yesexpr     "^[[:blank:]]*[JjYy][[:alpha:]]*"                                1
% noexpr      "^[[:blank:]]*[Nn][[:alpha:]]*"                                  1

yesexpr      "<'/>><<(><<(>:blank:<)/>><)/>>*<<(>JjYy<)/>>/                    1
<<(><<(>:alpha:<)/>><)/>>*"                                                    1
noexpr       "<'/>><<(><<(>:blank:<)/>><)/>>*<<(>Nn<)/>>/                      1
<<(><<(>:alpha:<)/>><)/>>*"                                                    1

END LC_MESSAGES
```

F.3.3 `kl_DK` — (Example) Greenlandic `LC_TIME` and `LC_MESSAGES`

```
escape_char /                                                                  1
comment_char %                                                                 1
% Danish example national locale for the Greenlandic language                 1
% Source: Danish Standards Association                                         1
% Revision: 1.7 1991-04-26                                                     1
%                                                                              1
% Only LC_TIME and LC_MESSAGES are specified here, else use the da_DK locale   1

LC_CTYPE
copy da_DK                                                                     1
END LC_CTYPE

LC_COLLATE                                                                     1
copy da_DK                                                                     1
END LC_COLLATE

LC_MONETARY                                                                    1
copy da_DK                                                                     1
END LC_MONETARY

LC_NUMERIC                                                                     1
copy da_DK                                                                     1
END LC_NUMERIC

LC_TIME                                                                        1

abday        "sab";"ata";"mar";"pin";"sis";"tal";"arf"                         1
day          "sabaat";"ataasinngorneq";"marlunngorneq";"pingasunngorneq";/     1
             "sisamanngorneq";"tallimanngorneq";"arfininngorneq"               1
abmon        "jan";"feb";"mar";"apr";"maj";"jun";/                             1
             "jul";"aug";"sep";"okt";"nov";"dec"                               1
mon          "januari";"februari";"martsi";"aprili";"maji";"juni";/            1
             "juli";"augustusi";"septemberi";"oktoberi";"novemberi";"decemberi" 1
d_t_fmt      "%a %d %b %Y %T %Z"                                               1
d_fmt        "%d %b %Y"                                                        1
```

```
t_fmt       "%T"                                                          1
am_pm       "";""
t_fmt_ampm  ""

END LC_TIME

LC_MESSAGES

% Must be careful to avoid interpreting "namik"/"nej" (no) as "aap"/"ja" (yes).  1

% yesexpr     "ˆ[[:blank:]]∗[JjYyAa][[:alpha:]]∗"                          1
% noexpr      "ˆ[[:blank:]]∗[Nn][[:alpha:]]∗"                              1

yesexpr     "<’/>><<(><<(>:blank:<)/>><)/>>∗<<(>JjYyAa<)/>>/               1
<<(><<(>:alpha:<)/>><)/>>∗"                                                1
noexpr      "<’/>><<(><<(>:blank:<)/>><)/>>∗<<(>Nn<)/>>/                   1
<<(><<(>:alpha:<)/>><)/>>∗"                                                1

END LC_MESSAGES
```

F.4 Character Mnemonics Guidelines

This clause presents guidelines for character mnemonics in a minimal coded character set. These guidelines are used within this sample annex and are recommended for other national profiles.

F.4.1 Aim of Character Mnemonics

The aim of the mnemonics is to be able to represent all characters in all standard coded character sets in any standard coded character set.

The usage of the character mnemonics is primarily intended within computer operating systems, programming languages, and applications and this work with character mnemonics is the current state of work that has been presented to the ISO working group responsible for these computer related issues, namely the ISO/IEC JTC 1/SC22 special working group on coded character set usage.

F.4.2 Covered Coded Character Sets

All characters in the standard coded character sets will be given a mnemonic to be represented in the minimal character set. The minimal coded character set is defined as the basic character set of ISO 646 {1}, where 12 positions are left undefined. The standard coded character sets are taken as the sum of all ISO-defined or ISO-registered coded character sets.

The most significant ISO coded character set is the ISO 10646 {B11} coded character set, whose aim is to code in 32 bits all characters in the world. These guidelines can be seen as assigning mnemonic attributes to most characters in ISO 10646 {B11}, currently at the DIS stage.

Other ISO coded character sets covered include all parts of ISO 8859 {B9} ISO 6937-2 {B6}, and all ISO 646 {1} conforming coded character sets in the ISO character set registry managed by ECMA according to ISO 4873 {4}. Some non-ISO coded character sets are also covered for convenience.

F.4.3 Character Mnemonics Classes

The character mnemonics are classified into two groups:

(1) A group with two-character mnemonics—Primarily intended for alphabetic scripts like Latin, Greek, Cyrillian, Hebrew, and Arabic, and special characters.

(2) A group with variable-length mnemonics—Primarily intended for nonalphabetic scripts like Japanese and Chinese. These mnemonics will have a unique lead-in and lead-out symbol.

All mnemonics are given a long descriptive name, written in the reference coded character set and taken from ISO 10646 {B11}, if possible.

F.4.4 Two-Character Mnemonics

The two-character mnemonics include various accented Latin letters, Greek, Cyrillic, Hebrew, Arabic, Hiragana, Katakana, and Bopomofo. Some special characters also are included. Almost all ISO or ISO-registered 7- and 8-bit coded character sets are covered with these two-character mnemonics.

The two characters are chosen so the graphical appearence in the reference set resembles as much as possible (within the possibilities available) the graphical appearance of the character. The basic coded character set of ISO 646 {1} is used as the reference set, as described previously.

The characters in the reference coded character set are chosen to represent themselves. They may be considered as two-character mnemonics where the second character is a space.

Control character mnemonics are chosen according to ISO 2047 {B3} and ISO 6429 {B5}.

Letters, including Greek, Cyrillic, Arabic, and Hebrew, are represented with the base letter as the first letter, and the second letter represents an accent or relation to a non-Latin script. Non-Latin letters are transliterated to Latin letters, following transliteration standards as closely as possible.

After a letter, the second character signifies the following:

exclamation-mark	!	grave
apostrophe	′	acute accent
greater-than-sign	>	circumflex accent

question-mark	`?`	tilde
hyphen-minus	`-`	macron
left-parenthesis	`(`	breve
full-stop	`.`	dot above/ring above
colon	`:`	diaeresis
comma	`,`	cedilla
underline	`_`	underline
solidus	`/`	stroke
quotation-mark	`"`	double acute accent
semicolon	`;`	ogonek
less-than-sign	`<`	caron
equals	`=`	Cyrillian
asterisk	`*`	Greek
percent-sign	`%`	Greek/Cyrillian special
plus	`+`	smalls: Arabic, capitals: Hebrew
four	`4`	Bopomofo
five	`5`	Hiragana
six	`6`	Katakana

Special characters are encoded with some mnemonic value. These are not systematic throughout, but most mnemonics start with a special character of the reference set. Special characters with some sort of reference to the reference coded character set normally have this character as the first character in the mnemonic.

F.4.5 Variable-Length Character Mnemonics

The variable-length character mnemonics are meant primarily for the ideographic characters in larger Asian coded character sets. To have the mnemonics as short as possible, which both saves storage and is easier to type, a short name is preferred. Considering the Chinese standard GB 2312 {B14} and the Japanese standards JIS X0208 {B15} and JIS X0212 {B16}, they are all given by row and column numbers between 1 and 99. So two positions for row and column and a coded character set identifier of one character would be almost as short as possible. The following coded character set identifiers are defined:

`c`	GB 2312 {B14}
`j`	JIS X0208 {B15}
`J`	JIS X0212 {B16}
`k`	KS C 5601 {B17}

The first idea was to have a name in Latin describing the pronunciation, but that is not possible according to Asian sources.

The variable-length character mnemonics can also be used for some Latin letters with more than one accent or other special characters that are used less frequently.

F.5 (Example) Danish Charmap Files

The (example) Danish locale is coded character-set independent, as it is defined in terms of symbolic character names. Symbolic character names are defined for about 1 300 characters, covering many coded character sets. It is not necessary to have all these characters present in the actual encoding character set because absent characters simply can be ignored. But specifying the locale with symbolic character names ensures a uniform collating sequence of the present characters, regardless of the encoded character set. The more complicated locale should not imply less efficient code at running time, although generating the locale tables could take a longer time.

Danish Standards provides several charmap files, of which the `ISO_10646` is the prime charmap, as it defines all the character names. It is expected, however, that the `ISO_8859-1` charmap would be of more current interest. The charmaps are quite general, and might be used for other countries' locales without change.

See the guidelines for character mnemonics in F.4 for guidance in reading these 1
charmap files.

F.5.1 ISO_10646 Charmap

```
# ISO/IEC DIS 10646: 1990 charmap based on ISO/IEC JTC1/SC2/WG2 N666
# Only a part of the 10646 encoding is tabled here

<escape_char> /
<mb_cur_max> 4
CHARMAP
<NUL>               /d000/d128/d128/d128    NULL (NUL)                          1
<SOH>               /d001/d128/d128/d128    START OF HEADING (SOH)              1
<STX>               /d002/d128/d128/d128    START OF TEXT (STX)                 1
<ETX>               /d003/d128/d128/d128    END OF TEXT (ETX)                   1
<EOT>               /d004/d128/d128/d128    END OF TRANSMISSION (EOT)           1
<ENQ>               /d005/d128/d128/d128    ENQUIRY (ENQ)                       1
<ACK>               /d006/d128/d128/d128    ACKNOWLEDGE (ACK)                   1
<alert>             /d007/d128/d128/d128    BELL (BEL)                          1
<BEL>               /d007/d128/d128/d128    BELL (BEL)                          1
<backspace>         /d008/d128/d128/d128    BACKSPACE (BS)                      1
<tab>               /d009/d128/d128/d128    CHARACTER TABULATION (HT)           1
<newline>           /d010/d128/d128/d128    LINE FEED (LF)                      1
<vertical-tab>      /d011/d128/d128/d128    LINE TABULATION (VT)                1
<form-feed>         /d012/d128/d128/d128    FORM FEED (FF)                      1
<carriage-return>   /d013/d128/d128/d128    CARRIAGE RETURN (CR)                1
<DLE>               /d016/d128/d128/d128    DATALINK ESCAPE (DLE)               1
<DC1>               /d017/d128/d128/d128    DEVICE CONTROL ONE (DC1)            1
<DC2>               /d018/d128/d128/d128    DEVICE CONTROL TWO (DC2)            1
<DC3>               /d019/d128/d128/d128    DEVICE CONTROL THREE (DC3)          1
<DC4>               /d020/d128/d128/d128    DEVICE CONTROL FOUR (DC4)           1
<NAK>               /d021/d128/d128/d128    NEGATIVE ACKNOWLEDGE (NAK)          1
<SYN>               /d022/d128/d128/d128    SYNCHRONOUS IDLE (SYN)              1
<ETB>               /d023/d128/d128/d128    END OF TRANSMISSION BLOCK (ETB)     1
<CAN>               /d024/d128/d128/d128    CANCEL (CAN)                        1
<SUB>               /d026/d128/d128/d128    SUBSTITUTE (SUB)                    1
<ESC>               /d027/d128/d128/d128    ESCAPE (ESC)                        1
```

```
<IS4>                   /d028/d128/d128/d128    FILE SEPARATOR (IS4)       1
<IS3>                   /d029/d128/d128/d128    GROUP SEPARATOR (IS3)      1
<intro>                 /d029/d128/d128/d128    GROUP SEPARATOR (IS3)      1
<IS2>                   /d030/d128/d128/d128    RECORD SEPARATOR (IS2)     1
<IS1>                   /d031/d128/d128/d128    UNIT SEPARATOR (IS1)       1
<DEL>                   /d127/d128/d128/d128    DELETE (DEL)               1
<space>                 /d032/d032/d032/d032    SPACE
<exclamation-mark>      /d032/d032/d032/d033    EXCLAMATION MARK
<quotation-mark>        /d032/d032/d032/d034    QUOTATION MARK
<number-sign>           /d032/d032/d032/d035    NUMBER SIGN
<dollar-sign>           /d032/d032/d032/d036    DOLLAR SIGN
<percent-sign>          /d032/d032/d032/d037    PERCENT SIGN
<ampersand>             /d032/d032/d032/d038    AMPERSAND
<apostrophe>            /d032/d032/d032/d039    APOSTROPHE
<left-parenthesis>      /d032/d032/d032/d040    LEFT PARENTHESIS
<right-parenthesis>     /d032/d032/d032/d041    RIGHT PARENTHESIS
<asterisk>              /d032/d032/d032/d042    ASTERISK
<plus-sign>             /d032/d032/d032/d043    PLUS SIGN
<comma>                 /d032/d032/d032/d044    COMMA
<hyphen>                /d032/d032/d032/d045    HYPHEN-MINUS
<hyphen-minus>          /d032/d032/d032/d045    HYPHEN-MINUS
<period>                /d032/d032/d032/d046    FULL STOP
<full-stop>             /d032/d032/d032/d046    FULL STOP
<slash>                 /d032/d032/d032/d047    SOLIDUS
<solidus>               /d032/d032/d032/d047    SOLIDUS
<zero>                  /d032/d032/d032/d048    DIGIT ZERO
<one>                   /d032/d032/d032/d049    DIGIT ONE
<two>                   /d032/d032/d032/d050    DIGIT TWO
<three>                 /d032/d032/d032/d051    DIGIT THREE
<four>                  /d032/d032/d032/d052    DIGIT FOUR
<five>                  /d032/d032/d032/d053    DIGIT FIVE
<six>                   /d032/d032/d032/d054    DIGIT SIX
<seven>                 /d032/d032/d032/d055    DIGIT SEVEN
<eight>                 /d032/d032/d032/d056    DIGIT EIGHT
<nine>                  /d032/d032/d032/d057    DIGIT NINE
<colon>                 /d032/d032/d032/d058    COLON
<semicolon>             /d032/d032/d032/d059    SEMICOLON
<less-than-sign>        /d032/d032/d032/d060    LESS-THAN SIGN
<equals-sign>           /d032/d032/d032/d061    EQUALS SIGN
<greater-than-sign>     /d032/d032/d032/d062    GREATER-THAN SIGN
<question-mark>         /d032/d032/d032/d063    QUESTION MARK
<commercial-at>         /d032/d032/d032/d064    COMMERCIAL AT
<left-square-bracket>   /d032/d032/d032/d091    LEFT SQUARE BRACKET
<reverse-solidus>       /d032/d032/d032/d092    REVERSE SOLIDUS
<backslash>             /d032/d032/d032/d092    REVERSE SOLIDUS
<right-square-bracket>  /d032/d032/d032/d093    RIGHT SQUARE BRACKET
<circumflex-accent>     /d032/d032/d032/d094    CIRCUMFLEX ACCENT
<low-line>              /d032/d032/d032/d095    LOW LINE
<underscore>            /d032/d032/d032/d095    LOW LINE
<grave-accent>          /d032/d032/d032/d096    GRAVE ACCENT
<left-curly-bracket>    /d032/d032/d032/d123    LEFT CURLY BRACKET
<vertical-line>         /d032/d032/d032/d124    VERTICAL LINE
<right-curly-bracket>   /d032/d032/d032/d125    RIGHT CURLY BRACKET
<tilde>                 /d032/d032/d032/d126    TILDE
<SP>    /d032/d032/d032/d032    SPACE
<!>     /d032/d032/d032/d033    EXCLAMATION MARK
<">     /d032/d032/d032/d034    QUOTATION MARK
```

```
<Nb>    /d032/d032/d032/d035    NUMBER SIGN
<DO>    /d032/d032/d032/d036    DOLLAR SIGN
<%>     /d032/d032/d032/d037    PERCENT SIGN
<&>     /d032/d032/d032/d038    AMPERSAND
<'>     /d032/d032/d032/d039    APOSTROPHE
<(>     /d032/d032/d032/d040    LEFT PARENTHESIS
<)>     /d032/d032/d032/d041    RIGHT PARENTHESIS
<*>     /d032/d032/d032/d042    ASTERISK
<+>     /d032/d032/d032/d043    PLUS SIGN
<,>     /d032/d032/d032/d044    COMMA
<->     /d032/d032/d032/d045    HYPHEN-MINUS
<.>     /d032/d032/d032/d046    FULL STOP
<//>    /d032/d032/d032/d047    SOLIDUS
<0>     /d032/d032/d032/d048    DIGIT ZERO
<1>     /d032/d032/d032/d049    DIGIT ONE
<2>     /d032/d032/d032/d050    DIGIT TWO
<3>     /d032/d032/d032/d051    DIGIT THREE
<4>     /d032/d032/d032/d052    DIGIT FOUR
<5>     /d032/d032/d032/d053    DIGIT FIVE
<6>     /d032/d032/d032/d054    DIGIT SIX
<7>     /d032/d032/d032/d055    DIGIT SEVEN
<8>     /d032/d032/d032/d056    DIGIT EIGHT
<9>     /d032/d032/d032/d057    DIGIT NINE
<:>     /d032/d032/d032/d058    COLON
<;>     /d032/d032/d032/d059    SEMICOLON
<<>     /d032/d032/d032/d060    LESS-THAN SIGN
<=>     /d032/d032/d032/d061    EQUALS SIGN
</>>    /d032/d032/d032/d062    GREATER-THAN SIGN
<?>     /d032/d032/d032/d063    QUESTION MARK
<At>    /d032/d032/d032/d064    COMMERCIAL AT
<A>     /d032/d032/d032/d065    LATIN CAPITAL LETTER A
<B>     /d032/d032/d032/d066    LATIN CAPITAL LETTER B
<C>     /d032/d032/d032/d067    LATIN CAPITAL LETTER C
<D>     /d032/d032/d032/d068    LATIN CAPITAL LETTER D
<E>     /d032/d032/d032/d069    LATIN CAPITAL LETTER E
<F>     /d032/d032/d032/d070    LATIN CAPITAL LETTER F
<G>     /d032/d032/d032/d071    LATIN CAPITAL LETTER G
<H>     /d032/d032/d032/d072    LATIN CAPITAL LETTER H
<I>     /d032/d032/d032/d073    LATIN CAPITAL LETTER I
<J>     /d032/d032/d032/d074    LATIN CAPITAL LETTER J
<K>     /d032/d032/d032/d075    LATIN CAPITAL LETTER K
<L>     /d032/d032/d032/d076    LATIN CAPITAL LETTER L
<M>     /d032/d032/d032/d077    LATIN CAPITAL LETTER M
<N>     /d032/d032/d032/d078    LATIN CAPITAL LETTER N
<O>     /d032/d032/d032/d079    LATIN CAPITAL LETTER O
<P>     /d032/d032/d032/d080    LATIN CAPITAL LETTER P
<Q>     /d032/d032/d032/d081    LATIN CAPITAL LETTER Q
<R>     /d032/d032/d032/d082    LATIN CAPITAL LETTER R
<S>     /d032/d032/d032/d083    LATIN CAPITAL LETTER S
<T>     /d032/d032/d032/d084    LATIN CAPITAL LETTER T
<U>     /d032/d032/d032/d085    LATIN CAPITAL LETTER U
<V>     /d032/d032/d032/d086    LATIN CAPITAL LETTER V
<W>     /d032/d032/d032/d087    LATIN CAPITAL LETTER W
<X>     /d032/d032/d032/d088    LATIN CAPITAL LETTER X
<Y>     /d032/d032/d032/d089    LATIN CAPITAL LETTER Y
<Z>     /d032/d032/d032/d090    LATIN CAPITAL LETTER Z
<<(>    /d032/d032/d032/d091    LEFT SQUARE BRACKET
```

```
<////>  /d032/d032/d032/d092    REVERSE SOLIDUS
<)/>>   /d032/d032/d032/d093    RIGHT SQUARE BRACKET
<'/>>   /d032/d032/d032/d094    CIRCUMFLEX ACCENT
<_>     /d032/d032/d032/d095    LOW LINE
<'!>    /d032/d032/d032/d096    GRAVE ACCENT
<a>     /d032/d032/d032/d097    LATIN SMALL LETTER A
<b>     /d032/d032/d032/d098    LATIN SMALL LETTER B
<c>     /d032/d032/d032/d099    LATIN SMALL LETTER C
<d>     /d032/d032/d032/d100    LATIN SMALL LETTER D
<e>     /d032/d032/d032/d101    LATIN SMALL LETTER E
<f>     /d032/d032/d032/d102    LATIN SMALL LETTER F
<g>     /d032/d032/d032/d103    LATIN SMALL LETTER G
<h>     /d032/d032/d032/d104    LATIN SMALL LETTER H
<i>     /d032/d032/d032/d105    LATIN SMALL LETTER I
<j>     /d032/d032/d032/d106    LATIN SMALL LETTER J
<k>     /d032/d032/d032/d107    LATIN SMALL LETTER K
<l>     /d032/d032/d032/d108    LATIN SMALL LETTER L
<m>     /d032/d032/d032/d109    LATIN SMALL LETTER M
<n>     /d032/d032/d032/d110    LATIN SMALL LETTER N
<o>     /d032/d032/d032/d111    LATIN SMALL LETTER O
<p>     /d032/d032/d032/d112    LATIN SMALL LETTER P
<q>     /d032/d032/d032/d113    LATIN SMALL LETTER Q
<r>     /d032/d032/d032/d114    LATIN SMALL LETTER R
<s>     /d032/d032/d032/d115    LATIN SMALL LETTER S
<t>     /d032/d032/d032/d116    LATIN SMALL LETTER T
<u>     /d032/d032/d032/d117    LATIN SMALL LETTER U
<v>     /d032/d032/d032/d118    LATIN SMALL LETTER V
<w>     /d032/d032/d032/d119    LATIN SMALL LETTER W
<x>     /d032/d032/d032/d120    LATIN SMALL LETTER X
<y>     /d032/d032/d032/d121    LATIN SMALL LETTER Y
<z>     /d032/d032/d032/d122    LATIN SMALL LETTER Z
<(!>    /d032/d032/d032/d123    LEFT CURLY BRACKET
<!!>    /d032/d032/d032/d124    VERTICAL LINE
<!)>    /d032/d032/d032/d125    RIGHT CURLY BRACKET
<'?>    /d032/d032/d032/d126    TILDE
<NS>    /d032/d032/d032/d160    NO-BREAK SPACE
<!I>    /d032/d032/d032/d161    INVERTED EXCLAMATION MARK
<Ct>    /d032/d032/d032/d162    CENT SIGN
<Pd>    /d032/d032/d032/d163    POUND SIGN
<Cu>    /d032/d032/d032/d164    CURRENCY SIGN
<Ye>    /d032/d032/d032/d165    YEN SIGN
<BB>    /d032/d032/d032/d166    BROKEN BAR
<SE>    /d032/d032/d032/d167    SECTION SIGN
<':>    /d032/d032/d032/d168    DIAERESIS
<Co>    /d032/d032/d032/d169    COPYRIGHT SIGN
<-a>    /d032/d032/d032/d170    FEMININE ORDINAL INDICATOR
<<<>    /d032/d032/d032/d171    LEFT POINTING DOUBLE ANGLE QUOTATION MARK
<NO>    /d032/d032/d032/d172    NOT SIGN
<-->    /d032/d032/d032/d173    SOFT HYPHEN
<Rg>    /d032/d032/d032/d174    REGISTERED SIGN
<'->    /d032/d032/d032/d175    MACRON
<DG>    /d032/d032/d032/d176    DEGREE SIGN
<+->    /d032/d032/d032/d177    PLUS-MINUS SIGN
<2S>    /d032/d032/d032/d178    SUPERSCRIPT TWO
<3S>    /d032/d032/d032/d179    SUPERSCRIPT THREE
<''>    /d032/d032/d032/d180    ACUTE ACCENT
<My>    /d032/d032/d032/d181    MICRO SIGN
```

```
<PI>     /d032/d032/d032/d182    PILCROW SIGN
<.M>     /d032/d032/d032/d183    MIDDLE DOT
<',>     /d032/d032/d032/d184    CEDILLA
<1S>     /d032/d032/d032/d185    SUPERSCRIPT ONE
<-o>     /d032/d032/d032/d186    MASCULINE ORDINAL INDICATOR
</>>>>   /d032/d032/d032/d187    RIGHT POINTING DOUBLE ANGLE QUOTATION MARK      1
<14>     /d032/d032/d032/d188    VULGAR FRACTION ONE QUARTER
<12>     /d032/d032/d032/d189    VULGAR FRACTION ONE HALF
<34>     /d032/d032/d032/d190    VULGAR FRACTION THREE QUARTERS
<?I>     /d032/d032/d032/d191    INVERTED QUESTION MARK
<A!>     /d032/d032/d032/d192    LATIN CAPITAL LETTER A WITH GRAVE
<A'>     /d032/d032/d032/d193    LATIN CAPITAL LETTER A WITH ACUTE
<A/>>    /d032/d032/d032/d194    LATIN CAPITAL LETTER A WITH CIRCUMFLEX
<A?>     /d032/d032/d032/d195    LATIN CAPITAL LETTER A WITH TILDE
<A:>     /d032/d032/d032/d196    LATIN CAPITAL LETTER A WITH DIAERESIS
<AA>     /d032/d032/d032/d197    LATIN CAPITAL LETTER A WITH RING ABOVE
<AE>     /d032/d032/d032/d198    LATIN CAPITAL LETTER AE
<C,>     /d032/d032/d032/d199    LATIN CAPITAL LETTER C WITH CEDILLA
<E!>     /d032/d032/d032/d200    LATIN CAPITAL LETTER E WITH GRAVE
<E'>     /d032/d032/d032/d201    LATIN CAPITAL LETTER E WITH ACUTE
<E/>>    /d032/d032/d032/d202    LATIN CAPITAL LETTER E WITH CIRCUMFLEX
<E:>     /d032/d032/d032/d203    LATIN CAPITAL LETTER E WITH DIAERESIS
<I!>     /d032/d032/d032/d204    LATIN CAPITAL LETTER I WITH GRAVE
<I'>     /d032/d032/d032/d205    LATIN CAPITAL LETTER I WITH ACUTE
<I/>>    /d032/d032/d032/d206    LATIN CAPITAL LETTER I WITH CIRCUMFLEX
<I:>     /d032/d032/d032/d207    LATIN CAPITAL LETTER I WITH DIAERESIS
<D->     /d032/d032/d032/d208    LATIN CAPITAL LETTER ETH (Icelandic)
<N?>     /d032/d032/d032/d209    LATIN CAPITAL LETTER N WITH TILDE
<O!>     /d032/d032/d032/d210    LATIN CAPITAL LETTER O WITH GRAVE
<O'>     /d032/d032/d032/d211    LATIN CAPITAL LETTER O WITH ACUTE
<O/>>    /d032/d032/d032/d212    LATIN CAPITAL LETTER O WITH CIRCUMFLEX
<O?>     /d032/d032/d032/d213    LATIN CAPITAL LETTER O WITH TILDE
<O:>     /d032/d032/d032/d214    LATIN CAPITAL LETTER O WITH DIAERESIS
<*X>     /d032/d032/d032/d215    MULTIPLICATION SIGN
<O//>    /d032/d032/d032/d216    LATIN CAPITAL LETTER O WITH STROKE
<U!>     /d032/d032/d032/d217    LATIN CAPITAL LETTER U WITH GRAVE
<U'>     /d032/d032/d032/d218    LATIN CAPITAL LETTER U WITH ACUTE
<U/>>    /d032/d032/d032/d219    LATIN CAPITAL LETTER U WITH CIRCUMFLEX
<U:>     /d032/d032/d032/d220    LATIN CAPITAL LETTER U WITH DIAERESIS
<Y'>     /d032/d032/d032/d221    LATIN CAPITAL LETTER Y WITH ACUTE
<TH>     /d032/d032/d032/d222    LATIN CAPITAL LETTER THORN (Icelandic)
<ss>     /d032/d032/d032/d223    LATIN SMALL LETTER SHARP S (German)
<a!>     /d032/d032/d032/d224    LATIN SMALL LETTER A WITH GRAVE
<a'>     /d032/d032/d032/d225    LATIN SMALL LETTER A WITH ACUTE
<a/>>    /d032/d032/d032/d226    LATIN SMALL LETTER A WITH CIRCUMFLEX
<a?>     /d032/d032/d032/d227    LATIN SMALL LETTER A WITH TILDE
<a:>     /d032/d032/d032/d228    LATIN SMALL LETTER A WITH DIAERESIS
<aa>     /d032/d032/d032/d229    LATIN SMALL LETTER A WITH RING ABOVE
<ae>     /d032/d032/d032/d230    LATIN SMALL LETTER AE
<c,>     /d032/d032/d032/d231    LATIN SMALL LETTER C WITH CEDILLA
<e!>     /d032/d032/d032/d232    LATIN SMALL LETTER E WITH GRAVE
<e'>     /d032/d032/d032/d233    LATIN SMALL LETTER E WITH ACUTE
<e/>>    /d032/d032/d032/d234    LATIN SMALL LETTER E WITH CIRCUMFLEX
<e:>     /d032/d032/d032/d235    LATIN SMALL LETTER E WITH DIAERESIS
<i!>     /d032/d032/d032/d236    LATIN SMALL LETTER I WITH GRAVE
<i'>     /d032/d032/d032/d237    LATIN SMALL LETTER I WITH ACUTE
<i/>>    /d032/d032/d032/d238    LATIN SMALL LETTER I WITH CIRCUMFLEX
```

```
<i:>     /d032/d032/d032/d239    LATIN SMALL LETTER I WITH DIAERESIS
<d->     /d032/d032/d032/d240    LATIN SMALL LETTER ETH (Icelandic)
<n?>     /d032/d032/d032/d241    LATIN SMALL LETTER N WITH TILDE
<o!>     /d032/d032/d032/d242    LATIN SMALL LETTER O WITH GRAVE
<o’>     /d032/d032/d032/d243    LATIN SMALL LETTER O WITH ACUTE
<o/>>    /d032/d032/d032/d244    LATIN SMALL LETTER O WITH CIRCUMFLEX
<o?>     /d032/d032/d032/d245    LATIN SMALL LETTER O WITH TILDE
<o:>     /d032/d032/d032/d246    LATIN SMALL LETTER O WITH DIAERESIS
<-:>     /d032/d032/d032/d247    DIVISION SIGN
<o//>    /d032/d032/d032/d248    LATIN SMALL LETTER O WITH STROKE
<u!>     /d032/d032/d032/d249    LATIN SMALL LETTER U WITH GRAVE
<u’>     /d032/d032/d032/d250    LATIN SMALL LETTER U WITH ACUTE
<u/>>    /d032/d032/d032/d251    LATIN SMALL LETTER U WITH CIRCUMFLEX
<u:>     /d032/d032/d032/d252    LATIN SMALL LETTER U WITH DIAERESIS
<y’>     /d032/d032/d032/d253    LATIN SMALL LETTER Y WITH ACUTE
<th>     /d032/d032/d032/d254    LATIN SMALL LETTER THORN (Icelandic)
<y:>     /d032/d032/d032/d255    LATIN SMALL LETTER Y WITH DIAERESIS
<A->     /d032/d032/d033/d033    LATIN CAPITAL LETTER A WITH MACRON
<C/>>    /d032/d032/d033/d034    LATIN CAPITAL LETTER C WITH CIRCUMFLEX
<C.>     /d032/d032/d033/d035    LATIN CAPITAL LETTER C WITH DOT ABOVE
<E->     /d032/d032/d033/d036    LATIN CAPITAL LETTER E WITH MACRON
<E.>     /d032/d032/d033/d037    LATIN CAPITAL LETTER E WITH DOT ABOVE
<G/>>    /d032/d032/d033/d039    LATIN CAPITAL LETTER G WITH CIRCUMFLEX
<’6>     /d032/d032/d033/d041    LEFT SINGLE QUOTATION MARK
<"6>     /d032/d032/d033/d042    LEFT DOUBLE QUOTATION MARK
<G(>     /d032/d032/d033/d043    LATIN CAPITAL LETTER G WITH BREVE
<<->     /d032/d032/d033/d044    LEFTWARD ARROW
<-!>     /d032/d032/d033/d045    UPWARD ARROW
<-/>>    /d032/d032/d033/d046    RIGHTWARD ARROW
<-v>     /d032/d032/d033/d047    DOWNWARD ARROW
<a->     /d032/d032/d033/d049    LATIN SMALL LETTER A WITH MACRON
<c/>>    /d032/d032/d033/d050    LATIN SMALL LETTER C WITH CIRCUMFLEX
<c.>     /d032/d032/d033/d051    LATIN SMALL LETTER C WITH DOT ABOVE
<e->     /d032/d032/d033/d052    LATIN SMALL LETTER E WITH MACRON
<e.>     /d032/d032/d033/d053    LATIN SMALL LETTER E WITH DOT ABOVE
<g/>>    /d032/d032/d033/d055    LATIN SMALL LETTER G WITH CIRCUMFLEX
<’9>     /d032/d032/d033/d057    RIGHT SINGLE QUOTATION MARK
<"9>     /d032/d032/d033/d058    RIGHT DOUBLE QUOTATION MARK
<g(>     /d032/d032/d033/d059    LATIN SMALL LETTER G WITH BREVE
<G.>     /d032/d032/d033/d065    LATIN CAPITAL LETTER G WITH DOT ABOVE
<G,>     /d032/d032/d033/d066    LATIN CAPITAL LETTER G WITH CEDILLA
<H/>>    /d032/d032/d033/d067    LATIN CAPITAL LETTER H WITH CIRCUMFLEX
<I?>     /d032/d032/d033/d070    LATIN CAPITAL LETTER I WITH TILDE
<I->     /d032/d032/d033/d071    LATIN CAPITAL LETTER I WITH MACRON
<I.>     /d032/d032/d033/d072    LATIN CAPITAL LETTER I WITH DOT ABOVE
<’0>     /d032/d032/d033/d074    RING ABOVE
<HB>     /d032/d032/d033/d080    HORIZONTAL BAR
<g.>     /d032/d032/d033/d081    LATIN SMALL LETTER G WITH DOT ABOVE
<g,>     /d032/d032/d033/d082    LATIN SMALL LETTER G WITH CEDILLA
<h/>>    /d032/d032/d033/d083    LATIN SMALL LETTER H WITH CIRCUMFLEX
<TM>     /d032/d032/d033/d084    TRADE MARK SIGN
<Md>     /d032/d032/d033/d085    MUSIC NOTE
<i?>     /d032/d032/d033/d086    LATIN SMALL LETTER I WITH TILDE
<i->     /d032/d032/d033/d087    LATIN SMALL LETTER I WITH MACRON
<18>     /d032/d032/d033/d092    VULGAR FRACTION ONE EIGHTH
<38>     /d032/d032/d033/d093    VULGAR FRACTION THREE EIGHTHS
<58>     /d032/d032/d033/d094    VULGAR FRACTION FIVE EIGHTHS
```

```
<78>    /d032/d032/d033/d095   VULGAR FRACTION SEVEN EIGHTHS
<Om>    /d032/d032/d033/d096   OHM SIGN
<I;>    /d032/d032/d033/d097   LATIN CAPITAL LETTER I WITH OGONEK
<J/>>   /d032/d032/d033/d098   LATIN CAPITAL LETTER J WITH CIRCUMFLEX
<K,>    /d032/d032/d033/d099   LATIN CAPITAL LETTER K WITH CEDILLA
<H//>   /d032/d032/d033/d100   LATIN CAPITAL LETTER H WITH STROKE
<IJ>    /d032/d032/d033/d102   LATIN CAPITAL LIGATURE IJ
<L.>    /d032/d032/d033/d103   LATIN CAPITAL LETTER L WITH MIDDLE DOT
<L,>    /d032/d032/d033/d104   LATIN CAPITAL LETTER L WITH CEDILLA
<N,>    /d032/d032/d033/d105   LATIN CAPITAL LETTER N WITH CEDILLA
<OE>    /d032/d032/d033/d106   LATIN CAPITAL LIGATURE OE
<O->    /d032/d032/d033/d107   LATIN CAPITAL LETTER O WITH MACRON
<T//>   /d032/d032/d033/d109   LATIN CAPITAL LETTER T WITH STROKE
<NG>    /d032/d032/d033/d110   LATIN CAPITAL LETTER ENG (Lappish)
<’n>    /d032/d032/d033/d111   LATIN SMALL LETTER N PRECEDED BY APOSTROPHE
<kk>    /d032/d032/d033/d112   LATIN SMALL LETTER KRA (Greenlandic)
<i;>    /d032/d032/d033/d113   LATIN SMALL LETTER I WITH OGONEK
<j/>>   /d032/d032/d033/d114   LATIN SMALL LETTER J WITH CIRCUMFLEX
<k,>    /d032/d032/d033/d115   LATIN SMALL LETTER K WITH CEDILLA
<h//>   /d032/d032/d033/d116   LATIN SMALL LETTER H WITH STROKE
<i.>    /d032/d032/d033/d117   LATIN SMALL LETTER I WITH NO DOT
<ij>    /d032/d032/d033/d118   LATIN SMALL LIGATURE IJ
<l.>    /d032/d032/d033/d119   LATIN SMALL LETTER L WITH MIDDLE DOT
<l,>    /d032/d032/d033/d120   LATIN SMALL LETTER L WITH CEDILLA
<n,>    /d032/d032/d033/d121   LATIN SMALL LETTER N WITH CEDILLA
<oe>    /d032/d032/d033/d122   LATIN SMALL LIGATURE OE
<o->    /d032/d032/d033/d123   LATIN SMALL LETTER O WITH MACRON
<t//>   /d032/d032/d033/d125   LATIN SMALL LETTER T WITH STROKE
<ng>    /d032/d032/d033/d126   LATIN SMALL LETTER ENG
<A;>    /d032/d032/d033/d161   LATIN CAPITAL LETTER A WITH OGONEK
<’(>    /d032/d032/d033/d162   BREVE
<L//>   /d032/d032/d033/d163   LATIN CAPITAL LETTER L WITH STROKE
<L<>    /d032/d032/d033/d165   LATIN CAPITAL LETTER L WITH CARON
<S’>    /d032/d032/d033/d166   LATIN CAPITAL LETTER S WITH ACUTE
<S/>>   /d032/d032/d033/d168   LATIN CAPITAL LETTER S WITH CIRCUMFLEX
<S<>    /d032/d032/d033/d169   LATIN CAPITAL LETTER S WITH CARON
<S,>    /d032/d032/d033/d170   LATIN CAPITAL LETTER S WITH CEDILLA
<T<>    /d032/d032/d033/d171   LATIN CAPITAL LETTER T WITH CARON
<Z’>    /d032/d032/d033/d172   LATIN CAPITAL LETTER Z WITH ACUTE
<Z<>    /d032/d032/d033/d174   LATIN CAPITAL LETTER Z WITH CARON
<Z.>    /d032/d032/d033/d175   LATIN CAPITAL LETTER Z WITH DOT ABOVE
<a;>    /d032/d032/d033/d177   LATIN SMALL LETTER A WITH OGONEK
<’;>    /d032/d032/d033/d178   OGONEK
<l//>   /d032/d032/d033/d179   LATIN SMALL LETTER L WITH STROKE
<l<>    /d032/d032/d033/d181   LATIN SMALL LETTER L WITH CARON
<s’>    /d032/d032/d033/d182   LATIN SMALL LETTER S WITH ACUTE
<’<>    /d032/d032/d033/d183   CARON
<s/>>   /d032/d032/d033/d184   LATIN SMALL LETTER S WITH CIRCUMFLEX
<s<>    /d032/d032/d033/d185   LATIN SMALL LETTER S WITH CARON
<s,>    /d032/d032/d033/d186   LATIN SMALL LETTER S WITH CEDILLA
<t<>    /d032/d032/d033/d187   LATIN SMALL LETTER T WITH CARON
<z’>    /d032/d032/d033/d188   LATIN SMALL LETTER Z WITH ACUTE
<’">    /d032/d032/d033/d189   DOUBLE ACUTE ACCENT
<z<>    /d032/d032/d033/d190   LATIN SMALL LETTER Z WITH CARON
<z.>    /d032/d032/d033/d191   LATIN SMALL LETTER Z WITH DOT ABOVE
<R’>    /d032/d032/d033/d192   LATIN CAPITAL LETTER R WITH ACUTE
<R,>    /d032/d032/d033/d193   LATIN CAPITAL LETTER R WITH CEDILLA
```

```
<A(>    /d032/d032/d033/d195    LATIN CAPITAL LETTER A WITH BREVE
<L’>    /d032/d032/d033/d197    LATIN CAPITAL LETTER L WITH ACUTE
<C’>    /d032/d032/d033/d198    LATIN CAPITAL LETTER C WITH ACUTE
<C<>    /d032/d032/d033/d200    LATIN CAPITAL LETTER C WITH CARON
<E;>    /d032/d032/d033/d202    LATIN CAPITAL LETTER E WITH OGONEK
<E<>    /d032/d032/d033/d204    LATIN CAPITAL LETTER E WITH CARON
<D<>    /d032/d032/d033/d207    LATIN CAPITAL LETTER D WITH CARON
<D//>   /d032/d032/d033/d208    LATIN CAPITAL LETTER D WITH STROKE
<N’>    /d032/d032/d033/d209    LATIN CAPITAL LETTER N WITH ACUTE
<N<>    /d032/d032/d033/d210    LATIN CAPITAL LETTER N WITH CARON
<U?>    /d032/d032/d033/d212    LATIN CAPITAL LETTER U WITH TILDE
<O">    /d032/d032/d033/d213    LATIN CAPITAL LETTER O WITH DOUBLE ACUTE
<U->    /d032/d032/d033/d214    LATIN CAPITAL LETTER U WITH MACRON
<U(>    /d032/d032/d033/d215    LATIN CAPITAL LETTER U WITH BREVE
<R<>    /d032/d032/d033/d216    LATIN CAPITAL LETTER R WITH CARON
<U0>    /d032/d032/d033/d217    LATIN CAPITAL LETTER U WITH RING ABOVE        1
<U;>    /d032/d032/d033/d218    LATIN CAPITAL LETTER U WITH OGONEK
<U">    /d032/d032/d033/d219    LATIN CAPITAL LETTER U WITH DOUBLE ACUTE
<W/>>   /d032/d032/d033/d220    LATIN CAPITAL LETTER W WITH CIRCUMFLEX
<Y/>>   /d032/d032/d033/d221    LATIN CAPITAL LETTER Y WITH CIRCUMFLEX
<T,>    /d032/d032/d033/d222    LATIN CAPITAL LETTER T WITH CEDILLA
<Y:>    /d032/d032/d033/d223    LATIN CAPITAL LETTER Y WITH DIAERESIS
<r’>    /d032/d032/d033/d224    LATIN SMALL LETTER R WITH ACUTE
<r,>    /d032/d032/d033/d225    LATIN SMALL LETTER R WITH CEDILLA
<a(>    /d032/d032/d033/d227    LATIN SMALL LETTER A WITH BREVE
<l’>    /d032/d032/d033/d229    LATIN SMALL LETTER L WITH ACUTE
<c’>    /d032/d032/d033/d230    LATIN SMALL LETTER C WITH ACUTE
<c<>    /d032/d032/d033/d232    LATIN SMALL LETTER C WITH CARON
<e;>    /d032/d032/d033/d234    LATIN SMALL LETTER E WITH OGONEK
<e<>    /d032/d032/d033/d236    LATIN SMALL LETTER E WITH CARON
<d<>    /d032/d032/d033/d239    LATIN SMALL LETTER D WITH CARON
<d//>   /d032/d032/d033/d240    LATIN SMALL LETTER D WITH STROKE
<n’>    /d032/d032/d033/d241    LATIN SMALL LETTER N WITH ACUTE
<n<>    /d032/d032/d033/d242    LATIN SMALL LETTER N WITH CARON
<u?>    /d032/d032/d033/d244    LATIN SMALL LETTER U WITH TILDE
<o">    /d032/d032/d033/d245    LATIN SMALL LETTER O WITH DOUBLE ACUTE
<u->    /d032/d032/d033/d246    LATIN SMALL LETTER U WITH MACRON
<u(>    /d032/d032/d033/d247    LATIN SMALL LETTER U WITH BREVE
<r<>    /d032/d032/d033/d248    LATIN SMALL LETTER R WITH CARON
<u0>    /d032/d032/d033/d249    LATIN SMALL LETTER U WITH RING ABOVE          1
<u;>    /d032/d032/d033/d250    LATIN SMALL LETTER U WITH OGONEK
<u">    /d032/d032/d033/d251    LATIN SMALL LETTER U WITH DOUBLE ACUTE
<w/>>   /d032/d032/d033/d252    LATIN SMALL LETTER W WITH CIRCUMFLEX
<y/>>   /d032/d032/d033/d253    LATIN SMALL LETTER Y WITH CIRCUMFLEX
<t,>    /d032/d032/d033/d254    LATIN SMALL LETTER T WITH CEDILLA
<’.>    /d032/d032/d033/d255    DOT ABOVE
<a<>    /d032/d032/d034/d032    LATIN SMALL LETTER A WITH CARON
<A<>    /d032/d032/d034/d033    LATIN CAPITAL LETTER A WITH CARON
<a_>    /d032/d032/d034/d034    LATIN SMALL LETTER A WITH LINE BELOW
<A_>    /d032/d032/d034/d035    LATIN CAPITAL LETTER A WITH LINE BELOW
<’a>    /d032/d032/d034/d048    LATIN SMALL LETTER A PRECEDED BY APOSTROPHE
<’A>    /d032/d032/d034/d049    LATIN CAPITAL LETTER A PRECEDED BY APOSTROPHE
<a1>    /d032/d032/d034/d052    LATIN SMALL LETTER A WITH MACRON AND DIAERESIS
<A1>    /d032/d032/d034/d053    LATIN CAPITAL LETTER A WITH MACRON AND DIAERESIS
<a2>    /d032/d032/d034/d054    LATIN SMALL LETTER A WITH MACRON AND DOT ABOVE
<A2>    /d032/d032/d034/d055    LATIN CAPITAL LETTER A WITH MACRON AND DOT ABOVE
<a3>    /d032/d032/d034/d056    LATIN SMALL LETTER AE WITH MACRON
```

```
<A3>    /d032/d032/d034/d057    LATIN CAPITAL LETTER AE WITH MACRON
<b.>    /d032/d032/d034/d086    LATIN SMALL LETTER B WITH DOT ABOVE
<B.>    /d032/d032/d034/d087    LATIN CAPITAL LETTER B WITH DOT ABOVE
<b_>    /d032/d032/d034/d088    LATIN SMALL LETTER B WITH LINE BELOW
<B_>    /d032/d032/d034/d089    LATIN CAPITAL LETTER B WITH LINE BELOW
<d_>    /d032/d032/d034/d096    LATIN SMALL LETTER D WITH LINE BELOW
<D_>    /d032/d032/d034/d097    LATIN CAPITAL LETTER D WITH LINE BELOW
<d.>    /d032/d032/d034/d098    LATIN SMALL LETTER D WITH DOT BELOW
<D.>    /d032/d032/d034/d099    LATIN CAPITAL LETTER D WITH DOT BELOW
<d;>    /d032/d032/d034/d100    LATIN SMALL LETTER D WITH OGONEK
<D;>    /d032/d032/d034/d101    LATIN CAPITAL LETTER D WITH OGONEK
<e(>    /d032/d032/d034/d106    LATIN SMALL LETTER E WITH BREVE
<E(>    /d032/d032/d034/d107    LATIN CAPITAL LETTER E WITH BREVE
<e_>    /d032/d032/d034/d108    LATIN SMALL LETTER E WITH LINE BELOW
<E_>    /d032/d032/d034/d109    LATIN CAPITAL LETTER E WITH LINE BELOW
<;S>    /d032/d032/d034/d126    HIGH OGONEK
<e?>    /d032/d032/d034/d168    LATIN SMALL LETTER E WITH TILDE
<E?>    /d032/d032/d034/d169    LATIN CAPITAL LETTER E WITH TILDE
<f.>    /d032/d032/d034/d180    LATIN SMALL LETTER F WITH DOT ABOVE
<F.>    /d032/d032/d034/d181    LATIN CAPITAL LETTER F WITH DOT ABOVE
<g<>    /d032/d032/d034/d182    LATIN SMALL LETTER G WITH CARON
<G<>    /d032/d032/d034/d183    LATIN CAPITAL LETTER G WITH CARON
<g->    /d032/d032/d034/d184    LATIN SMALL LETTER G WITH MACRON
<G->    /d032/d032/d034/d185    LATIN CAPITAL LETTER G WITH MACRON
<g//>   /d032/d032/d034/d188    LATIN SMALL LETTER G WITH STROKE
<G//>   /d032/d032/d034/d189    LATIN CAPITAL LETTER G WITH STROKE
<h:>    /d032/d032/d034/d192    LATIN SMALL LETTER H WITH DIAERESIS
<H:>    /d032/d032/d034/d193    LATIN CAPITAL LETTER H WITH DIAERESIS
<h.>    /d032/d032/d034/d194    LATIN SMALL LETTER H WITH DOT ABOVE
<H.>    /d032/d032/d034/d195    LATIN CAPITAL LETTER H WITH DOT ABOVE
<h,>    /d032/d032/d034/d196    LATIN SMALL LETTER H WITH CEDILLA
<H,>    /d032/d032/d034/d197    LATIN CAPITAL LETTER H WITH CEDILLA
<h;>    /d032/d032/d034/d198    LATIN SMALL LETTER H WITH OGONEK
<H;>    /d032/d032/d034/d199    LATIN CAPITAL LETTER H WITH OGONEK
<i<>    /d032/d032/d034/d204    LATIN SMALL LETTER I WITH CARON
<I<>    /d032/d032/d034/d205    LATIN CAPITAL LETTER I WITH CARON
<i(>    /d032/d032/d034/d206    LATIN SMALL LETTER I WITH BREVE
<I(>    /d032/d032/d034/d207    LATIN CAPITAL LETTER I WITH BREVE
<j(>    /d032/d032/d034/d224    LATIN SMALL LETTER J WITH BREVE
<J(>    /d032/d032/d034/d225    LATIN CAPITAL LETTER J WITH BREVE
<k’>    /d032/d032/d034/d226    LATIN SMALL LETTER K WITH ACUTE
<K’>    /d032/d032/d034/d227    LATIN CAPITAL LETTER K WITH ACUTE
<k<>    /d032/d032/d034/d228    LATIN SMALL LETTER K WITH CARON
<K<>    /d032/d032/d034/d229    LATIN CAPITAL LETTER K WITH CARON
<k_>    /d032/d032/d034/d230    LATIN SMALL LETTER K WITH LINE BELOW
<K_>    /d032/d032/d034/d231    LATIN CAPITAL LETTER K WITH LINE BELOW
<k.>    /d032/d032/d034/d232    LATIN SMALL LETTER K WITH DOT BELOW
<K.>    /d032/d032/d034/d233    LATIN CAPITAL LETTER K WITH DOT BELOW
<k;>    /d032/d032/d034/d234    LATIN SMALL LETTER K WITH OGONEK
<K;>    /d032/d032/d034/d235    LATIN CAPITAL LETTER K WITH OGONEK
<l_>    /d032/d032/d034/d240    LATIN SMALL LETTER L WITH LINE BELOW
<L_>    /d032/d032/d034/d241    LATIN CAPITAL LETTER L WITH LINE BELOW
<m’>    /d032/d032/d034/d248    LATIN SMALL LETTER M WITH ACUTE
<M’>    /d032/d032/d034/d249    LATIN CAPITAL LETTER M WITH ACUTE
<m.>    /d032/d032/d034/d250    LATIN SMALL LETTER M WITH DOT ABOVE
<M.>    /d032/d032/d034/d251    LATIN CAPITAL LETTER M WITH DOT ABOVE
<n.>    /d032/d032/d035/d034    LATIN SMALL LETTER N WITH DOT ABOVE
```

```
<N.>    /d032/d032/d035/d035    LATIN CAPITAL LETTER N WITH DOT ABOVE
<n_>    /d032/d032/d035/d038    LATIN SMALL LETTER N WITH LINE BELOW
<N_>    /d032/d032/d035/d039    LATIN CAPITAL LETTER N WITH LINE BELOW
<o<>    /d032/d032/d035/d046    LATIN SMALL LETTER O WITH CARON
<O<>    /d032/d032/d035/d047    LATIN CAPITAL LETTER O WITH CARON
<o(>    /d032/d032/d035/d048    LATIN SMALL LETTER O WITH BREVE
<O(>    /d032/d032/d035/d049    LATIN CAPITAL LETTER O WITH BREVE
<o_>    /d032/d032/d035/d050    LATIN SMALL LETTER O WITH LINE BELOW
<O_>    /d032/d032/d035/d051    LATIN CAPITAL LETTER O WITH LINE BELOW
<o;>    /d032/d032/d035/d064    LATIN SMALL LETTER O WITH OGONEK
<O;>    /d032/d032/d035/d065    LATIN CAPITAL LETTER O WITH OGONEK
<o1>    /d032/d032/d035/d068    LATIN SMALL LETTER O WITH MACRON AND OGONEK
<O1>    /d032/d032/d035/d069    LATIN CAPITAL LETTER O WITH MACRON AND OGONEK
<p'>    /d032/d032/d035/d098    LATIN SMALL LETTER P WITH ACUTE
<P'>    /d032/d032/d035/d099    LATIN CAPITAL LETTER P WITH ACUTE
<r.>    /d032/d032/d035/d100    LATIN SMALL LETTER R WITH DOT ABOVE
<R.>    /d032/d032/d035/d101    LATIN CAPITAL LETTER R WITH DOT ABOVE
<r_>    /d032/d032/d035/d102    LATIN SMALL LETTER R WITH LINE BELOW
<R_>    /d032/d032/d035/d103    LATIN CAPITAL LETTER R WITH LINE BELOW
<s.>    /d032/d032/d035/d110    LATIN SMALL LETTER S WITH DOT ABOVE
<S.>    /d032/d032/d035/d111    LATIN CAPITAL LETTER S WITH DOT ABOVE
<s;>    /d032/d032/d035/d114    LATIN SMALL LETTER S WITH OGONEK
<S;>    /d032/d032/d035/d115    LATIN CAPITAL LETTER S WITH OGONEK
<t_>    /d032/d032/d035/d160    LATIN SMALL LETTER T WITH LINE BELOW
<T_>    /d032/d032/d035/d161    LATIN CAPITAL LETTER T WITH LINE BELOW
<t.>    /d032/d032/d035/d162    LATIN SMALL LETTER T WITH DOT BELOW
<T.>    /d032/d032/d035/d163    LATIN CAPITAL LETTER T WITH DOT BELOW
<u<>    /d032/d032/d035/d170    LATIN SMALL LETTER U WITH CARON
<U<>    /d032/d032/d035/d171    LATIN CAPITAL LETTER U WITH CARON
<v?>    /d032/d032/d035/d214    LATIN SMALL LETTER V WITH TILDE
<V?>    /d032/d032/d035/d215    LATIN CAPITAL LETTER V WITH TILDE
<w'>    /d032/d032/d035/d220    LATIN SMALL LETTER W WITH ACUTE
<W'>    /d032/d032/d035/d221    LATIN CAPITAL LETTER W WITH ACUTE
<w.>    /d032/d032/d035/d222    LATIN SMALL LETTER W WITH DOT ABOVE
<W.>    /d032/d032/d035/d223    LATIN CAPITAL LETTER W WITH DOT ABOVE
<w:>    /d032/d032/d035/d224    LATIN SMALL LETTER W WITH DIAERESIS
<W:>    /d032/d032/d035/d225    LATIN CAPITAL LETTER W WITH DIAERESIS
<x.>    /d032/d032/d035/d230    LATIN SMALL LETTER X WITH DOT ABOVE
<X.>    /d032/d032/d035/d231    LATIN CAPITAL LETTER X WITH DOT ABOVE
<x:>    /d032/d032/d035/d232    LATIN SMALL LETTER X WITH DIAERESIS
<X:>    /d032/d032/d035/d233    LATIN CAPITAL LETTER X WITH DIAERESIS
<y!>    /d032/d032/d035/d236    LATIN SMALL LETTER Y WITH GRAVE
<Y!>    /d032/d032/d035/d237    LATIN CAPITAL LETTER Y WITH GRAVE
<y.>    /d032/d032/d035/d238    LATIN SMALL LETTER Y WITH DOT ABOVE
<Y.>    /d032/d032/d035/d239    LATIN CAPITAL LETTER Y WITH DOT ABOVE
<z/>>   /d032/d032/d035/d244    LATIN SMALL LETTER Z WITH CIRCUMFLEX
<Z/>>   /d032/d032/d035/d245    LATIN CAPITAL LETTER Z WITH CIRCUMFLEX
<z(>    /d032/d032/d035/d246    LATIN SMALL LETTER Z WITH BREVE
<Z(>    /d032/d032/d035/d247    LATIN CAPITAL LETTER Z WITH BREVE
<z_>    /d032/d032/d035/d248    LATIN SMALL LETTER Z WITH LINE BELOW
<Z_>    /d032/d032/d035/d249    LATIN CAPITAL LETTER Z WITH LINE BELOW
<z//>   /d032/d032/d035/d252    LATIN SMALL LETTER Z WITH STROKE
<Z//>   /d032/d032/d035/d253    LATIN CAPITAL LETTER Z WITH STROKE
<ez>    /d032/d032/d035/d254    LATIN SMALL LETTER EZH WITH CARON
<EZ>    /d032/d032/d035/d255    LATIN CAPITAL LETTER EZH WITH CARON
<g'>    /d032/d032/d036/d033    LATIN SMALL LETTER G WITH ACUTE
<G'>    /d032/d032/d036/d034    LATIN CAPITAL LETTER G WITH ACUTE
```

```
<'b>    /d032/d032/d036/d084    LATIN SMALL LETTER B PRECEDED BY APOSTROPHE
<'B>    /d032/d032/d036/d085    LATIN CAPITAL LETTER B PRECEDED BY APOSTROPHE
<'d>    /d032/d032/d036/d096    LATIN SMALL LETTER D PRECEDED BY APOSTROPHE
<'D>    /d032/d032/d036/d097    LATIN CAPITAL LETTER D PRECEDED BY APOSTROPHE
<'g>    /d032/d032/d036/d162    LATIN SMALL LETTER G PRECEDED BY APOSTROPHE
<'G>    /d032/d032/d036/d163    LATIN CAPITAL LETTER G PRECEDED BY APOSTROPHE
<'j>    /d032/d032/d036/d174    LATIN SMALL LETTER J PRECEDED BY APOSTROPHE
<'J>    /d032/d032/d036/d175    LATIN CAPITAL LETTER J PRECEDED BY APOSTROPHE
<'y>    /d032/d032/d036/d235    LATIN SMALL LETTER Y PRECEDED BY APOSTROPHE
<'Y>    /d032/d032/d036/d236    LATIN CAPITAL LETTER Y PRECEDED BY APOSTROPHE
<ed>    /d032/d032/d036/d239    LATIN SMALL LETTER EDZ
<ED>    /d032/d032/d036/d240    LATIN CAPITAL LETTER EDZ
<Vs>    /d032/d032/d037/d032    SPACE SYMBOL
<1M>    /d032/d032/d037/d033    EM-SPACE
<1N>    /d032/d032/d037/d034    EN-SPACE
<3M>    /d032/d032/d037/d035    THREE-PER-EM SPACE
<4M>    /d032/d032/d037/d036    FOUR-PER-EM SPACE
<6M>    /d032/d032/d037/d037    SIX-PER-EM SPACE
<1H>    /d032/d032/d037/d038    HAIR SPACE
<1T>    /d032/d032/d037/d039    THIN SPACE
<-1>    /d032/d032/d037/d040    HYPHEN
<-N>    /d032/d032/d037/d041    EN-DASH
<-2>    /d032/d032/d037/d042    MINUS SIGN
<-M>    /d032/d032/d037/d043    EM-DASH
<-3>    /d032/d032/d037/d044    QUOTATION DASH
<'1>    /d032/d032/d037/d045    SINGLE PRIME
<'2>    /d032/d032/d037/d046    DOUBLE PRIME
<'3>    /d032/d032/d037/d047    TRIPLE PRIME
<9'>    /d032/d032/d037/d048    SINGLE HIGH-REVERSED-9 QUOTATION MARK
<9">    /d032/d032/d037/d049    DOUBLE HIGH-REVERSED-9 QUOTATION MARK
<.9>    /d032/d032/d037/d050    SINGLE LOW-9 QUOTATION MARK
<:9>    /d032/d032/d037/d051    DOUBLE LOW-9 QUOTATION MARK
<<1>    /d032/d032/d037/d052    SINGLE LEFT-POINTING ANGLE QUOTATION MARK
</>1>   /d032/d032/d037/d053    SINGLE RIGHT-POINTING ANGLE QUOTATION MARK
<<//>   /d032/d032/d037/d054    LEFT-POINTING ANGLE BRACKET
<///>>  /d032/d032/d037/d055    RIGHT-POINTING ANGLE BRACKET
<15>    /d032/d032/d037/d056    VULGAR FRACTION ONE FIFTH
<25>    /d032/d032/d037/d057    VULGAR FRACTION TWO FIFTHS
<35>    /d032/d032/d037/d058    VULGAR FRACTION THREE FIFTHS
<45>    /d032/d032/d037/d059    VULGAR FRACTION FOUR FIFTHS
<16>    /d032/d032/d037/d060    VULGAR FRACTION ONE SIXTH
<13>    /d032/d032/d037/d061    VULGAR FRACTION ONE THIRD
<23>    /d032/d032/d037/d062    VULGAR FRACTION TWO THIRDS
<56>    /d032/d032/d037/d063    VULGAR FRACTION FIVE SIXTHS
<*->    /d032/d032/d037/d064    MIDDLE ASTERISK
<//->   /d032/d032/d037/d065    DAGGER
<//=>   /d032/d032/d037/d066    DOUBLE-DAGGER
<-X>    /d032/d032/d037/d067    MALTESE CROSS
<%0>    /d032/d032/d037/d068    PER-MILLE SIGN
<co>    /d032/d032/d037/d069    CARE-OF SIGN
<PO>    /d032/d032/d037/d070    SOUND RECORDING COPYRIGHT SIGN
<Rx>    /d032/d032/d037/d071    PRESCRIPTION SIGN
<AO>    /d032/d032/d037/d072    ANGSTROEM SIGN
<oC>    /d032/d032/d037/d073    CENTIGRADE DEGREE SIGN
<Ml>    /d032/d032/d037/d074    MALE SIGN
<Fm>    /d032/d032/d037/d075    FEMALE SIGN
<Tl>    /d032/d032/d037/d076    TELEPHONE SIGN
```

<TR>	/d032/d032/d037/d077	TELEPHONE RECORDER SIGN
<MX>	/d032/d032/d037/d078	MUSICAL SHARP SIGN
<Mb>	/d032/d032/d037/d079	MUSICAL FLAT SIGN
<Mx>	/d032/d032/d037/d080	MUSICAL NATURAL SIGN
<XX>	/d032/d032/d037/d081	BALLOT CROSS SIGN
<OK>	/d032/d032/d037/d082	CHECK MARK
<M2>	/d032/d032/d037/d083	DOUBLE MUSICAL NOTES
<!2>	/d032/d032/d037/d084	DOUBLE EXCLAMATION MARKS
<=2>	/d032/d032/d037/d085	DOUBLE LOW LINE
<Ca>	/d032/d032/d037/d086	CARET
<..>	/d032/d032/d037/d087	TWO-DOT LEADER
<.3>	/d032/d032/d037/d088	HORIZONTAL ELLIPSIS
<:3>	/d032/d032/d037/d089	VERTICAL ELLIPSIS
<.:>	/d032/d032/d037/d090	THEREFORE SIGN
<:.>	/d032/d032/d037/d091	BECAUSE SIGN
<-+>	/d032/d032/d037/d092	MINUS-PLUS SIGN
<!=>	/d032/d032/d037/d093	NOT EQUAL-TO SIGN
<=3>	/d032/d032/d037/d094	IDENTICAL-TO SIGN
<?1>	/d032/d032/d037/d095	DIFFERENCE-BETWEEN SIGN
<?2>	/d032/d032/d037/d096	ALMOST-EQUALS SIGN
<?->	/d032/d032/d037/d097	ASYMTOTICALLY-EQUALS SIGN
<?=>	/d032/d032/d037/d098	SIMILAR-TO SIGN
<=<>	/d032/d032/d037/d099	LESS-THAN OR EQUAL-TO SIGN
</>=>	/d032/d032/d037/d100	GREATER-THAN OR EQUAL-TO SIGN
<0(>	/d032/d032/d037/d101	PROPORTIONAL-TO SIGN
<00>	/d032/d032/d037/d102	INFINITY SIGN
<PP>	/d032/d032/d037/d103	PARALLEL-TO SIGN
<-T>	/d032/d032/d037/d104	ORTHOGONAL-TO SIGN
<-L>	/d032/d032/d037/d105	RIGHT ANGLE SIGN
<-V>	/d032/d032/d037/d106	ANGLE SIGN
<AN>	/d032/d032/d037/d107	LOGICAL-AND SIGN
<OR>	/d032/d032/d037/d108	LOGICAL-OR SIGN
<.P>	/d032/d032/d037/d109	PRODUCT DOT SIGN
<nS>	/d032/d032/d037/d110	SUPERSCRIPT LATIN SMALL LETTER N
<dP>	/d032/d032/d037/d111	PARTIAL DIFFERENTIAL SIGN
<f(>	/d032/d032/d037/d112	FUNCTION SIGN
<In>	/d032/d032/d037/d113	INTEGRAL SIGN
<Io>	/d032/d032/d037/d114	CONTOUR INTEGRAL SIGN
<RT>	/d032/d032/d037/d117	RADICAL SIGN
<*P>	/d032/d032/d037/d118	REPEATED PRODUCT SIGN
<+Z>	/d032/d032/d037/d119	SUMMATION SIGN
<FA>	/d032/d032/d037/d120	FOR-ALL SIGN
<TE>	/d032/d032/d037/d121	THERE-EXISTS SIGN
<GF>	/d032/d032/d037/d122	GAMMA FUNCTION SIGN
<DE>	/d032/d032/d037/d123	INCREMENT SIGN
<NB>	/d032/d032/d037/d124	NABLA
<(U>	/d032/d032/d037/d125	INTERSECTION SIGN
<)U>	/d032/d032/d037/d126	UNION SIGN
<(C>	/d032/d032/d037/d160	PROPER SUBSET SIGN
<)C>	/d032/d032/d037/d161	PROPER SUPERSET SIGN
<(_>	/d032/d032/d037/d162	SUBSET SIGN
<)_>	/d032/d032/d037/d163	SUPERSET SIGN
<(->	/d032/d032/d037/d164	ELEMENT-OF SIGN
<-)>	/d032/d032/d037/d165	HAS AN ELEMENT SIGN
<</>>	/d032/d032/d037/d166	LEFT AND RIGHT-POINTING ARROW
<UD>	/d032/d032/d037/d167	UP AND DOWN-POINTING ARROW
<Ub>	/d032/d032/d037/d168	UP AND DOWN-POINTING ARROW WITH LINE BELOW

```
<<=>     /d032/d032/d037/d169    IMPLIED-BY SIGN
<=/>>    /d032/d032/d037/d170    IMPLIES SIGN
<==>     /d032/d032/d037/d171    IF-AND-ONLY-IF SIGN
<//0>    /d032/d032/d037/d172    EMPTY SIGN
<OL>     /d032/d032/d037/d173    SOLID LOZENGE
<0u>     /d032/d032/d037/d176    SMILING FACE WHITE
<0U>     /d032/d032/d037/d177    SMILING FACE BLACK
<SU>     /d032/d032/d037/d178    RADIANT SUN
<0:>     /d032/d032/d037/d179    DOTTED CIRCLE
<OS>     /d032/d032/d037/d180    SQUARE EMPTY
<fS>     /d032/d032/d037/d181    SQUARE SOLID
<Or>     /d032/d032/d037/d182    RECTANGLE EMPTY
<SR>     /d032/d032/d037/d183    RECTANGLE SOLID
<uT>     /d032/d032/d037/d184    UPWARDS-POINTING TRIANGLE EMPTY
<UT>     /d032/d032/d037/d185    UPWARDS-POINTING TRIANGLE SOLID
<dT>     /d032/d032/d037/d186    DOWNWARDS-POINTING TRIANGLE EMPTY
<Dt>     /d032/d032/d037/d187    DOWNWARDS-POINTING TRIANGLE SOLID
<PL>     /d032/d032/d037/d188    LEFTWARDS POINTER SOLID
<PR>     /d032/d032/d037/d189    RIGHTWARDS POINTER SOLID
<*1>     /d032/d032/d037/d190    STAR EMPTY
<*2>     /d032/d032/d037/d191    STAR SOLID
<VV>     /d032/d032/d037/d192    BOX DRAWINGS HEAVY VERTICAL
<HH>     /d032/d032/d037/d193    BOX DRAWINGS HEAVY HORIZONTAL
<DR>     /d032/d032/d037/d194    BOX DRAWINGS HEAVY DOWN AND RIGHT
<LD>     /d032/d032/d037/d195    BOX DRAWINGS HEAVY DOWN AND LEFT
<UR>     /d032/d032/d037/d196    BOX DRAWINGS HEAVY UP AND RIGHT
<UL>     /d032/d032/d037/d197    BOX DRAWINGS HEAVY UP AND LEFT
<VR>     /d032/d032/d037/d198    BOX DRAWINGS HEAVY VERTICAL AND RIGHT
<VL>     /d032/d032/d037/d199    BOX DRAWINGS HEAVY VERTICAL AND LEFT
<DH>     /d032/d032/d037/d200    BOX DRAWINGS HEAVY HORIZONTAL AND DOWN
<UH>     /d032/d032/d037/d201    BOX DRAWINGS HEAVY HORIZONTAL AND UP
<VH>     /d032/d032/d037/d202    BOX DRAWINGS HEAVY VERTICAL AND HORIZONTAL
<TB>     /d032/d032/d037/d203    BOX DRAWING SOLID UPPER HALF BLOCK
<LB>     /d032/d032/d037/d204    BOX DRAWING SOLID LOWER HALF BLOCK
<FB>     /d032/d032/d037/d205    BOX DRAWING SOLID FULL BLOCK
<sB>     /d032/d032/d037/d206    BOX DRAWING SOLID SMALL SQUARE
<EH>     /d032/d032/d037/d207    EMPTY HOUSE SIGN
<vv>     /d032/d032/d037/d208    BOX DRAWINGS LIGHT VERTICAL
<hh>     /d032/d032/d037/d209    BOX DRAWINGS LIGHT HORIZONTAL
<dr>     /d032/d032/d037/d210    BOX DRAWINGS LIGHT DOWN AND RIGHT
<dl>     /d032/d032/d037/d211    BOX DRAWINGS LIGHT DOWN AND LEFT
<ur>     /d032/d032/d037/d212    BOX DRAWINGS LIGHT UP AND RIGHT
<ul>     /d032/d032/d037/d213    BOX DRAWINGS LIGHT UP AND LEFT
<vr>     /d032/d032/d037/d214    BOX DRAWINGS LIGHT VERTICAL AND RIGHT
<vl>     /d032/d032/d037/d215    BOX DRAWINGS LIGHT VERTICAL AND LEFT
<dh>     /d032/d032/d037/d216    BOX DRAWINGS LIGHT HORIZONTAL AND DOWN
<uh>     /d032/d032/d037/d217    BOX DRAWINGS LIGHT HORIZONTAL AND UP
<vh>     /d032/d032/d037/d218    BOX DRAWINGS LIGHT VERTICAL AND HORIZONTAL
<.S>     /d032/d032/d037/d219    BOX DRAWING LIGHT SHADE (25%)
<:S>     /d032/d032/d037/d220    BOX DRAWING MEDIUM SHADE (50%)
<?S>     /d032/d032/d037/d221    BOX DRAWING DARK SHADE (75%)
<lB>     /d032/d032/d037/d222    BOX DRAWING SOLID LEFT HALF BLOCK
<RB>     /d032/d032/d037/d223    BOX DRAWING SOLID RIGHT HALF BLOCK
<cC>     /d032/d032/d037/d224    CLUB SYMBOL
<cD>     /d032/d032/d037/d225    DIAMOND SYMBOL
<Dr>     /d032/d032/d037/d226    BOX DRAWINGS DOWN HEAVY AND RIGHT LIGHT
<Dl>     /d032/d032/d037/d227    BOX DRAWINGS DOWN HEAVY AND LEFT LIGHT
```

```
<Ur>     /d032/d032/d037/d228    BOX DRAWINGS UP HEAVY AND RIGHT LIGHT
<Ul>     /d032/d032/d037/d229    BOX DRAWINGS UP HEAVY AND LEFT LIGHT
<Vr>     /d032/d032/d037/d230    BOX DRAWINGS VERTICAL HEAVY AND RIGHT LIGHT
<Vl>     /d032/d032/d037/d231    BOX DRAWINGS VERTICAL HEAVY AND LEFT LIGHT
<dH>     /d032/d032/d037/d232    BOX DRAWINGS HORIZONTAL HEAVY AND DOWN LIGHT
<uH>     /d032/d032/d037/d233    BOX DRAWINGS HORIZONTAL HEAVY AND UP LIGHT
<vH>     /d032/d032/d037/d234    BOX DRAWINGS VERTICAL LIGHT AND HORIZONTAL HEAVY
<Ob>     /d032/d032/d037/d235    CIRCLE BULLET EMPTY
<Sb>     /d032/d032/d037/d236    CIRCLE BULLET SOLID
<Sn>     /d032/d032/d037/d237    CIRCLE BULLET NEGATIVE
<Pt>     /d032/d032/d037/d238    PESETA SYMBOL
<NI>     /d032/d032/d037/d239    REVERSED NOT SIGN
<cH>     /d032/d032/d037/d240    HEART SYMBOL
<cS>     /d032/d032/d037/d241    SPADE SYMBOL
<dR>     /d032/d032/d037/d242    BOX DRAWINGS DOWN LIGHT AND RIGHT HEAVY
<dL>     /d032/d032/d037/d243    BOX DRAWINGS DOWN LIGHT AND LEFT HEAVY
<uR>     /d032/d032/d037/d244    BOX DRAWINGS UP LIGHT AND RIGHT HEAVY
<uL>     /d032/d032/d037/d245    BOX DRAWINGS UP LIGHT AND LEFT HEAVY
<vR>     /d032/d032/d037/d246    BOX DRAWINGS VERTICAL LIGHT AND RIGHT HEAVY
<vL>     /d032/d032/d037/d247    BOX DRAWINGS VERTICAL LIGHT AND LEFT HEAVY
<Dh>     /d032/d032/d037/d248    BOX DRAWINGS HORIZONTAL LIGHT AND DOWN HEAVY
<Uh>     /d032/d032/d037/d249    BOX DRAWINGS HORIZONTAL LIGHT AND UP HEAVY
<Vh>     /d032/d032/d037/d250    BOX DRAWINGS VERTICAL HEAVY AND HORIZONTAL LIGHT
<0m>     /d032/d032/d037/d251    MEDIUM CIRCLE EMPTY
<0M>     /d032/d032/d037/d252    MEDIUM CIRCLE SOLID
<Ic>     /d032/d032/d037/d253    MEDIUM CIRCLE NEGATIVE
<SM>     /d032/d032/d037/d254    SERVICE MARK SIGN
<CG>     /d032/d032/d037/d255    CONGRUENCE SIGN
<Ci>     /d032/d032/d038/d037    CIRCLE
<(A>     /d032/d032/d038/d041    ARC SIGN
</>V>    /d032/d032/d038/d046    RIGHTWARDS VECTOR ABOVE
<!<>     /d032/d032/d038/d049    NOT LESS-THAN SIGN
<<*>     /d032/d032/d038/d056    MUCH-LESS-THAN SIGN
<!/>>    /d032/d032/d038/d065    NOT GREATER-THAN SIGN
<*/>>    /d032/d032/d038/d072    MUCH-GREATER-THAN SIGN
<<7>     /d032/d032/d038/d094    CEILING SIGN LEFT
<7<>     /d032/d032/d038/d095    FLOOR SIGN LEFT
</>7>    /d032/d032/d038/d110    CEILING SIGN RIGHT
<7/>>    /d032/d032/d038/d111    FLOOR SIGN RIGHT
<I2>     /d032/d032/d038/d121    DOUBLE INTEGRAL SIGN
<0.>     /d032/d032/d038/d164    DOT IN RING
<HI>     /d032/d032/d038/d177    HAS-AN-IMAGE SIGN
<::>     /d032/d032/d038/d193    PROPORTION SIGN
<FD>     /d032/d032/d038/d209    FORWARD DIAGONAL
<LZ>     /d032/d032/d038/d223    LOZENGE
<BD>     /d032/d032/d038/d225    BACKWARD DIAGONAL
<1R>     /d032/d032/d039/d032    ROMAN NUMERAL ONE
<2R>     /d032/d032/d039/d033    ROMAN NUMERAL TWO
<3R>     /d032/d032/d039/d034    ROMAN NUMERAL THREE
<4R>     /d032/d032/d039/d035    ROMAN NUMERAL FOUR
<5R>     /d032/d032/d039/d036    ROMAN NUMERAL FIVE
<6R>     /d032/d032/d039/d037    ROMAN NUMERAL SIX
<7R>     /d032/d032/d039/d038    ROMAN NUMERAL SEVEN
<8R>     /d032/d032/d039/d039    ROMAN NUMERAL EIGHT
<9R>     /d032/d032/d039/d040    ROMAN NUMERAL NINE
<aR>     /d032/d032/d039/d041    ROMAN NUMERAL TEN
<bR>     /d032/d032/d039/d042    ROMAN NUMERAL ELEVEN
```

```
<cR>     /d032/d032/d039/d043    ROMAN NUMERAL TWELVE
<IO>     /d032/d032/d040/d161    CYRILLIC CAPITAL LETTER IO
<D%>     /d032/d032/d040/d162    CYRILLIC CAPITAL LETTER DJE (Serbocroatian)
<G%>     /d032/d032/d040/d163    CYRILLIC CAPITAL LETTER GJE (Macedonian)
<IE>     /d032/d032/d040/d164    CYRILLIC CAPITAL LETTER UKRAINIAN IE
<DS>     /d032/d032/d040/d165    CYRILLIC CAPITAL LETTER DZE (Macedonian)
<II>     /d032/d032/d040/d166    CYRILLIC CAPITAL LETTER BYELORUSSIAN-UKRAINIAN I
<YI>     /d032/d032/d040/d167    CYRILLIC CAPITAL LETTER YI (Ukrainian)
<J%>     /d032/d032/d040/d168    CYRILLIC CAPITAL LETTER JE
<LJ>     /d032/d032/d040/d169    CYRILLIC CAPITAL LETTER LJE
<NJ>     /d032/d032/d040/d170    CYRILLIC CAPITAL LETTER NJE
<Ts>     /d032/d032/d040/d171    CYRILLIC CAPITAL LETTER TSHE (Serbocroatian)
<KJ>     /d032/d032/d040/d172    CYRILLIC CAPITAL LETTER KJE (Macedonian)
<V%>     /d032/d032/d040/d174    CYRILLIC CAPITAL LETTER SHORT U (Byelorussian)
<DZ>     /d032/d032/d040/d175    CYRILLIC CAPITAL LETTER DZHE
<A=>     /d032/d032/d040/d176    CYRILLIC CAPITAL LETTER A
<B=>     /d032/d032/d040/d177    CYRILLIC CAPITAL LETTER BE
<V=>     /d032/d032/d040/d178    CYRILLIC CAPITAL LETTER VE
<G=>     /d032/d032/d040/d179    CYRILLIC CAPITAL LETTER GHE
<D=>     /d032/d032/d040/d180    CYRILLIC CAPITAL LETTER DE
<E=>     /d032/d032/d040/d181    CYRILLIC CAPITAL LETTER IE
<Z%>     /d032/d032/d040/d182    CYRILLIC CAPITAL LETTER ZHE
<Z=>     /d032/d032/d040/d183    CYRILLIC CAPITAL LETTER ZE
<I=>     /d032/d032/d040/d184    CYRILLIC CAPITAL LETTER I
<J=>     /d032/d032/d040/d185    CYRILLIC CAPITAL LETTER SHORT I
<K=>     /d032/d032/d040/d186    CYRILLIC CAPITAL LETTER KA
<L=>     /d032/d032/d040/d187    CYRILLIC CAPITAL LETTER EL
<M=>     /d032/d032/d040/d188    CYRILLIC CAPITAL LETTER EM
<N=>     /d032/d032/d040/d189    CYRILLIC CAPITAL LETTER EN
<O=>     /d032/d032/d040/d190    CYRILLIC CAPITAL LETTER O
<P=>     /d032/d032/d040/d191    CYRILLIC CAPITAL LETTER PE
<R=>     /d032/d032/d040/d192    CYRILLIC CAPITAL LETTER ER
<S=>     /d032/d032/d040/d193    CYRILLIC CAPITAL LETTER ES
<T=>     /d032/d032/d040/d194    CYRILLIC CAPITAL LETTER TE
<U=>     /d032/d032/d040/d195    CYRILLIC CAPITAL LETTER U
<F=>     /d032/d032/d040/d196    CYRILLIC CAPITAL LETTER EF
<H=>     /d032/d032/d040/d197    CYRILLIC CAPITAL LETTER HA
<C=>     /d032/d032/d040/d198    CYRILLIC CAPITAL LETTER TSE
<C%>     /d032/d032/d040/d199    CYRILLIC CAPITAL LETTER CHE
<S%>     /d032/d032/d040/d200    CYRILLIC CAPITAL LETTER SHA
<Sc>     /d032/d032/d040/d201    CYRILLIC CAPITAL LETTER SHCHA
<=">     /d032/d032/d040/d202    CYRILLIC CAPITAL HARD SIGN
<Y=>     /d032/d032/d040/d203    CYRILLIC CAPITAL LETTER YERU
<%">     /d032/d032/d040/d204    CYRILLIC CAPITAL SOFT SIGN
<JE>     /d032/d032/d040/d205    CYRILLIC CAPITAL LETTER E
<JU>     /d032/d032/d040/d206    CYRILLIC CAPITAL LETTER YU
<JA>     /d032/d032/d040/d207    CYRILLIC CAPITAL LETTER YA
<a=>     /d032/d032/d040/d208    CYRILLIC SMALL LETTER A
<b=>     /d032/d032/d040/d209    CYRILLIC SMALL LETTER BE
<v=>     /d032/d032/d040/d210    CYRILLIC SMALL LETTER VE
<g=>     /d032/d032/d040/d211    CYRILLIC SMALL LETTER GHE
<d=>     /d032/d032/d040/d212    CYRILLIC SMALL LETTER DE
<e=>     /d032/d032/d040/d213    CYRILLIC SMALL LETTER IE
<z%>     /d032/d032/d040/d214    CYRILLIC SMALL LETTER ZHE
<z=>     /d032/d032/d040/d215    CYRILLIC SMALL LETTER ZE
<i=>     /d032/d032/d040/d216    CYRILLIC SMALL LETTER I
<j=>     /d032/d032/d040/d217    CYRILLIC SMALL LETTER SHORT I
```

```
<k=>    /d032/d032/d040/d218    CYRILLIC SMALL LETTER KA
<l=>    /d032/d032/d040/d219    CYRILLIC SMALL LETTER EL
<m=>    /d032/d032/d040/d220    CYRILLIC SMALL LETTER EM
<n=>    /d032/d032/d040/d221    CYRILLIC SMALL LETTER EN
<o=>    /d032/d032/d040/d222    CYRILLIC SMALL LETTER O
<p=>    /d032/d032/d040/d223    CYRILLIC SMALL LETTER PE
<r=>    /d032/d032/d040/d224    CYRILLIC SMALL LETTER ER
<s=>    /d032/d032/d040/d225    CYRILLIC SMALL LETTER ES
<t=>    /d032/d032/d040/d226    CYRILLIC SMALL LETTER TE
<u=>    /d032/d032/d040/d227    CYRILLIC SMALL LETTER U
<f=>    /d032/d032/d040/d228    CYRILLIC SMALL LETTER EF
<h=>    /d032/d032/d040/d229    CYRILLIC SMALL LETTER HA
<c=>    /d032/d032/d040/d230    CYRILLIC SMALL LETTER TSE
<c%>    /d032/d032/d040/d231    CYRILLIC SMALL LETTER CHE
<s%>    /d032/d032/d040/d232    CYRILLIC SMALL LETTER SHA
<sc>    /d032/d032/d040/d233    CYRILLIC SMALL LETTER SHCHA
<='>    /d032/d032/d040/d234    CYRILLIC SMALL HARD SIGN
<y=>    /d032/d032/d040/d235    CYRILLIC SMALL LETTER YERU
<%'>    /d032/d032/d040/d236    CYRILLIC SMALL SOFT SIGN
<je>    /d032/d032/d040/d237    CYRILLIC SMALL LETTER E
<ju>    /d032/d032/d040/d238    CYRILLIC SMALL LETTER YU
<ja>    /d032/d032/d040/d239    CYRILLIC SMALL LETTER YA
<N0>    /d032/d032/d040/d240    NUMERO SIGN
<io>    /d032/d032/d040/d241    CYRILLIC SMALL LETTER IO
<d%>    /d032/d032/d040/d242    CYRILLIC SMALL LETTER DJE (Serbocroatian)
<g%>    /d032/d032/d040/d243    CYRILLIC SMALL LETTER GJE (Macedonian)
<ie>    /d032/d032/d040/d244    CYRILLIC SMALL LETTER UKRAINIAN IE
<ds>    /d032/d032/d040/d245    CYRILLIC SMALL LETTER DZE (Macedonian)
<ii>    /d032/d032/d040/d246    CYRILLIC SMALL LETTER BYELORUSSIAN-UKRAINIAN I
<yi>    /d032/d032/d040/d247    CYRILLIC SMALL LETTER YI (Ukrainian)
<j%>    /d032/d032/d040/d248    CYRILLIC SMALL LETTER JE
<lj>    /d032/d032/d040/d249    CYRILLIC SMALL LETTER LJE
<nj>    /d032/d032/d040/d250    CYRILLIC SMALL LETTER NJE
<ts>    /d032/d032/d040/d251    CYRILLIC SMALL LETTER TSHE (Serbocroatian)
<kj>    /d032/d032/d040/d252    CYRILLIC SMALL LETTER KJE (Macedonian)
<v%>    /d032/d032/d040/d254    CYRILLIC SMALL LETTER SHORT U (Byelorussian)
<dz>    /d032/d032/d040/d255    CYRILLIC SMALL LETTER DZHE
<i3>    /d032/d032/d042/d160    GREEK IOTA BELOW
<;;>    /d032/d032/d042/d161    GREEK DAISA PNEUMATA (rough)
<,,>    /d032/d032/d042/d162    GREEK PSILI PNEUMATA (smooth)
<!*>    /d032/d032/d042/d164    GREEK VARIA
<?*>    /d032/d032/d042/d165    GREEK PERISPOMENI
<;'>    /d032/d032/d042/d166    GREEK DAISA AND ACUTE ACCENT
<,'>    /d032/d032/d042/d167    GREEK PSILI AND ACUTE ACCENT
<;!>    /d032/d032/d042/d168    GREEK DAISA AND VARIA
<,!>    /d032/d032/d042/d169    GREEK PSILI AND VARIA
<?;>    /d032/d032/d042/d170    GREEK PERISPOMENI AND DAISA
<?,>    /d032/d032/d042/d171    GREEK PERISPOMENI AND PSILI
<!:>    /d032/d032/d042/d174    GREEK VARIA AND DIAERESIS
<?:>    /d032/d032/d042/d175    GREEK PERISPOMENI AND DIAERESIS
<I3>    /d032/d032/d042/d176    GREEK CAPITAL LETTER IOTA WITH PERISPOMENI
#                               AND PSILI
<'%>    /d032/d032/d042/d181    ACUTE ACCENT AND DIAERESIS (Tonos and Dialytica)
<A%>    /d032/d032/d042/d182    GREEK CAPITAL LETTER ALPHA WITH ACUTE
<E%>    /d032/d032/d042/d184    GREEK CAPITAL LETTER EPSILON WITH ACUTE
<Y%>    /d032/d032/d042/d185    GREEK CAPITAL LETTER ETA WITH ACUTE
<I%>    /d032/d032/d042/d186    GREEK CAPITAL LETTER IOTA WITH ACUTE
```

```
<O%>    /d032/d032/d042/d188    GREEK CAPITAL LETTER OMICRON WITH ACUTE
<U%>    /d032/d032/d042/d190    GREEK CAPITAL LETTER UPSILON WITH ACUTE
<W%>    /d032/d032/d042/d191    GREEK CAPITAL LETTER OMEGA WITH ACUTE
<A*>    /d032/d032/d042/d193    GREEK CAPITAL LETTER ALPHA
<B*>    /d032/d032/d042/d194    GREEK CAPITAL LETTER BETA
<G*>    /d032/d032/d042/d195    GREEK CAPITAL LETTER GAMMA
<D*>    /d032/d032/d042/d196    GREEK CAPITAL LETTER DELTA
<E*>    /d032/d032/d042/d197    GREEK CAPITAL LETTER EPSILON
<Z*>    /d032/d032/d042/d198    GREEK CAPITAL LETTER ZETA
<Y*>    /d032/d032/d042/d199    GREEK CAPITAL LETTER ETA
<H*>    /d032/d032/d042/d200    GREEK CAPITAL LETTER THETA
<I*>    /d032/d032/d042/d201    GREEK CAPITAL LETTER IOTA
<K*>    /d032/d032/d042/d202    GREEK CAPITAL LETTER KAPPA
<L*>    /d032/d032/d042/d203    GREEK CAPITAL LETTER LAMDA
<M*>    /d032/d032/d042/d204    GREEK CAPITAL LETTER MU
<N*>    /d032/d032/d042/d205    GREEK CAPITAL LETTER NU
<C*>    /d032/d032/d042/d206    GREEK CAPITAL LETTER XI
<O*>    /d032/d032/d042/d207    GREEK CAPITAL LETTER OMICRON
<P*>    /d032/d032/d042/d208    GREEK CAPITAL LETTER PI
<R*>    /d032/d032/d042/d209    GREEK CAPITAL LETTER RHO
<S*>    /d032/d032/d042/d211    GREEK CAPITAL LETTER SIGMA
<T*>    /d032/d032/d042/d212    GREEK CAPITAL LETTER TAU
<U*>    /d032/d032/d042/d213    GREEK CAPITAL LETTER UPSILON
<F*>    /d032/d032/d042/d214    GREEK CAPITAL LETTER PHI
<X*>    /d032/d032/d042/d215    GREEK CAPITAL LETTER CHI
<Q*>    /d032/d032/d042/d216    GREEK CAPITAL LETTER PSI
<W*>    /d032/d032/d042/d217    GREEK CAPITAL LETTER OMEGA
<J*>    /d032/d032/d042/d218    GREEK CAPITAL LETTER IOTA WITH DIAERESIS
<V*>    /d032/d032/d042/d219    GREEK CAPITAL LETTER UPSILON WITH DIAERESIS
<a%>    /d032/d032/d042/d220    GREEK SMALL LETTER ALPHA WITH ACUTE
<e%>    /d032/d032/d042/d221    GREEK SMALL LETTER EPSILON WITH ACUTE
<y%>    /d032/d032/d042/d222    GREEK SMALL LETTER ETA WITH ACUTE
<i%>    /d032/d032/d042/d223    GREEK SMALL LETTER IOTA WITH ACUTE
<a*>    /d032/d032/d042/d225    GREEK SMALL LETTER ALPHA
<b*>    /d032/d032/d042/d226    GREEK SMALL LETTER BETA
<g*>    /d032/d032/d042/d227    GREEK SMALL LETTER GAMMA
<d*>    /d032/d032/d042/d228    GREEK SMALL LETTER DELTA
<e*>    /d032/d032/d042/d229    GREEK SMALL LETTER EPSILON
<z*>    /d032/d032/d042/d230    GREEK SMALL LETTER ZETA
<y*>    /d032/d032/d042/d231    GREEK SMALL LETTER ETA
<h*>    /d032/d032/d042/d232    GREEK SMALL LETTER THETA
<i*>    /d032/d032/d042/d233    GREEK SMALL LETTER IOTA
<k*>    /d032/d032/d042/d234    GREEK SMALL LETTER KAPPA
<l*>    /d032/d032/d042/d235    GREEK SMALL LETTER LAMDA
<m*>    /d032/d032/d042/d236    GREEK SMALL LETTER MU
<n*>    /d032/d032/d042/d237    GREEK SMALL LETTER NU
<c*>    /d032/d032/d042/d238    GREEK SMALL LETTER XI
<o*>    /d032/d032/d042/d239    GREEK SMALL LETTER OMICRON
<p*>    /d032/d032/d042/d240    GREEK SMALL LETTER PI
<r*>    /d032/d032/d042/d241    GREEK SMALL LETTER RHO
<*s>    /d032/d032/d042/d242    GREEK SMALL LETTER FINAL SIGMA
<s*>    /d032/d032/d042/d243    GREEK SMALL LETTER SIGMA
<t*>    /d032/d032/d042/d244    GREEK SMALL LETTER TAU
<u*>    /d032/d032/d042/d245    GREEK SMALL LETTER UPSILON
<f*>    /d032/d032/d042/d246    GREEK SMALL LETTER PHI
<x*>    /d032/d032/d042/d247    GREEK SMALL LETTER CHI
<q*>    /d032/d032/d042/d248    GREEK SMALL LETTER PSI
```

```
<w*>    /d032/d032/d042/d249    GREEK SMALL LETTER OMEGA
<j*>    /d032/d032/d042/d250    GREEK SMALL LETTER IOTA WITH DIAERESIS
<v*>    /d032/d032/d042/d251    GREEK SMALL LETTER UPSILON WITH DIAERESIS
<o%>    /d032/d032/d042/d252    GREEK SMALL LETTER OMICRON WITH ACUTE
<u%>    /d032/d032/d042/d253    GREEK SMALL LETTER UPSILON WITH ACUTE
<w%>    /d032/d032/d042/d254    GREEK SMALL LETTER OMEGA WITH ACUTE
<p+>    /d032/d032/d044/d035    ARABIC LETTER PEH
<v+>    /d032/d032/d044/d040    ARABIC LETTER VEH
<gf>    /d032/d032/d044/d052    ARABIC LETTER GAF
<,+>    /d032/d032/d044/d172    ARABIC COMMA
<;+>    /d032/d032/d044/d187    ARABIC SEMICOLON
<?+>    /d032/d032/d044/d191    ARABIC QUESTION MARK
<H'>    /d032/d032/d044/d193    ARABIC LETTER HAMZA
<aM>    /d032/d032/d044/d194    ARABIC LETTER ALEF WITH MADDA ABOVE
<aH>    /d032/d032/d044/d195    ARABIC LETTER ALEF WITH HAMZA ABOVE
<wH>    /d032/d032/d044/d196    ARABIC LETTER WAW WITH HAMZA ABOVE
<ah>    /d032/d032/d044/d197    ARABIC LETTER ALEF WITH HAMZA BELOW
<yH>    /d032/d032/d044/d198    ARABIC LETTER YEH WITH HAMZA ABOVE
<a+>    /d032/d032/d044/d199    ARABIC LETTER ALEF
<b+>    /d032/d032/d044/d200    ARABIC LETTER BEH
<tm>    /d032/d032/d044/d201    ARABIC LETTER TEH MARBUTA
<t+>    /d032/d032/d044/d202    ARABIC LETTER TEH
<tk>    /d032/d032/d044/d203    ARABIC LETTER THEH
<g+>    /d032/d032/d044/d204    ARABIC LETTER JEEM
<hk>    /d032/d032/d044/d205    ARABIC LETTER HAH
<x+>    /d032/d032/d044/d206    ARABIC LETTER KHAH
<d+>    /d032/d032/d044/d207    ARABIC LETTER DAL
<dk>    /d032/d032/d044/d208    ARABIC LETTER THAL
<r+>    /d032/d032/d044/d209    ARABIC LETTER RA
<z+>    /d032/d032/d044/d210    ARABIC LETTER ZAIN
<s+>    /d032/d032/d044/d211    ARABIC LETTER SEEN
<sn>    /d032/d032/d044/d212    ARABIC LETTER SHEEN
<c+>    /d032/d032/d044/d213    ARABIC LETTER SAD
<dd>    /d032/d032/d044/d214    ARABIC LETTER DAD
<tj>    /d032/d032/d044/d215    ARABIC LETTER TAH
<zH>    /d032/d032/d044/d216    ARABIC LETTER ZAH
<e+>    /d032/d032/d044/d217    ARABIC LETTER AIN
<i+>    /d032/d032/d044/d218    ARABIC LETTER GHAIN
<++>    /d032/d032/d044/d224    ARABIC TATWEEL
<f+>    /d032/d032/d044/d225    ARABIC LETTER FEH
<q+>    /d032/d032/d044/d226    ARABIC LETTER QAF
<k+>    /d032/d032/d044/d227    ARABIC LETTER KAF
<l+>    /d032/d032/d044/d228    ARABIC LETTER LAM
<m+>    /d032/d032/d044/d229    ARABIC LETTER MEEM
<n+>    /d032/d032/d044/d230    ARABIC LETTER NOON
<h+>    /d032/d032/d044/d231    ARABIC LETTER HEH
<w+>    /d032/d032/d044/d232    ARABIC LETTER WAW
<j+>    /d032/d032/d044/d233    ARABIC LETTER ALEF MAKSURA
<y+>    /d032/d032/d044/d234    ARABIC LETTER YEH
<:+>    /d032/d032/d044/d235    ARABIC FATHATAN
<"+>    /d032/d032/d044/d236    ARABIC DAMMATAN
<=+>    /d032/d032/d044/d237    ARABIC KASRATAN
<//+>   /d032/d032/d044/d238    ARABIC FATHA
<'+>    /d032/d032/d044/d239    ARABIC DAMMA
<1+>    /d032/d032/d044/d240    ARABIC KASRA
<3+>    /d032/d032/d044/d241    ARABIC SHADDA
<0+>    /d032/d032/d044/d242    ARABIC SUKUN
```

```
<A+>     /d032/d032/d045/d224    HEBREW LETTER ALEF
<B+>     /d032/d032/d045/d225    HEBREW LETTER BET
<G+>     /d032/d032/d045/d226    HEBREW LETTER GIMEL
<D+>     /d032/d032/d045/d227    HEBREW LETTER DALET
<H+>     /d032/d032/d045/d228    HEBREW LETTER HE
<W+>     /d032/d032/d045/d229    HEBREW LETTER VAV
<Z+>     /d032/d032/d045/d230    HEBREW LETTER ZAYIN
<X+>     /d032/d032/d045/d231    HEBREW LETTER HET
<Tj>     /d032/d032/d045/d232    HEBREW LETTER TET
<J+>     /d032/d032/d045/d233    HEBREW LETTER YOD
<K%>     /d032/d032/d045/d234    HEBREW LETTER FINAL KAF
<K+>     /d032/d032/d045/d235    HEBREW LETTER KAF
<L+>     /d032/d032/d045/d236    HEBREW LETTER LAMED
<M%>     /d032/d032/d045/d237    HEBREW LETTER FINAL MEM
<M+>     /d032/d032/d045/d238    HEBREW LETTER MEM
<N%>     /d032/d032/d045/d239    HEBREW LETTER FINAL NUN
<N+>     /d032/d032/d045/d240    HEBREW LETTER NUN
<S+>     /d032/d032/d045/d241    HEBREW LETTER SAMEKH
<E+>     /d032/d032/d045/d242    HEBREW LETTER AYIN
<P%>     /d032/d032/d045/d243    HEBREW LETTER FINAL PE
<P+>     /d032/d032/d045/d244    HEBREW LETTER PE
<Zj>     /d032/d032/d045/d245    HEBREW LETTER FINAL TSADI
<ZJ>     /d032/d032/d045/d246    HEBREW LETTER TSADI
<Q+>     /d032/d032/d045/d247    HEBREW LETTER QOF
<R+>     /d032/d032/d045/d248    HEBREW LETTER RESH
<Sh>     /d032/d032/d045/d249    HEBREW LETTER SIN
<T+>     /d032/d032/d045/d250    HEBREW LETTER TAV
<IS>     /d032/d032/d046/d032    IDEOGRAPHIC SPACE
<,_>     /d032/d032/d046/d033    IDEOGRAPHIC COMMA
<._>     /d032/d032/d046/d034    IDEOGRAPHIC FULL STOP
<+">     /d032/d032/d046/d035    DITTO MARK
<+_>     /d032/d032/d046/d036    IDEOGRAPHIC DITTO MARK
<*_>     /d032/d032/d046/d037    IDEOGRAPHIC REPETITION MARK
<;_>     /d032/d032/d046/d038    IDEOGRAPHIC CLOSING MARK
<0_>     /d032/d032/d046/d039    IDEOGRAPHIC NUMBER ZERO
<<+>     /d032/d032/d046/d042    LEFT-POINTING DOUBLE ANGLE BRACKET
</>+>    /d032/d032/d046/d043    RIGHT-POINTING DOUBLE ANGLE BRACKET
<<’>     /d032/d032/d046/d044    IDEOGRAPHIC LEFT BRACKET
</>’>    /d032/d032/d046/d045    IDEOGRAPHIC RIGHT BRACKET
<<">     /d032/d032/d046/d046    IDEOGRAPHIC LEFT DOUBLE BRACKET
</>">    /d032/d032/d046/d047    IDEOGRAPHIC RIGHT DOUBLE BRACKET
<(">     /d032/d032/d046/d048    LEFT BOLDFACE SQUARE BRACKET
<)">     /d032/d032/d046/d049    RIGHT BOLDFACE SQUARE BRACKET
<=//>    /d032/d032/d046/d050    POSTAL MARK
<=_>     /d032/d032/d046/d051    GETA MARK
<(’>     /d032/d032/d046/d052    LEFT TORTOISE-SHELL BRACKET
<)’>     /d032/d032/d046/d053    RIGHT TORTOISE-SHELL BRACKET
<KM>     /d032/d032/d046/d054    KOME MARK
<b4>     /d032/d032/d046/d069    BOPOMOFO LETTER B
<p4>     /d032/d032/d046/d070    BOPOMOFO LETTER P
<m4>     /d032/d032/d046/d071    BOPOMOFO LETTER M
<f4>     /d032/d032/d046/d072    BOPOMOFO LETTER F
<d4>     /d032/d032/d046/d073    BOPOMOFO LETTER D
<t4>     /d032/d032/d046/d074    BOPOMOFO LETTER T
<n4>     /d032/d032/d046/d075    BOPOMOFO LETTER N
<l4>     /d032/d032/d046/d076    BOPOMOFO LETTER L
<g4>     /d032/d032/d046/d077    BOPOMOFO LETTER G
```

```
<k4>     /d032/d032/d046/d078    BOPOMOFO LETTER K
<h4>     /d032/d032/d046/d079    BOPOMOFO LETTER H
<j4>     /d032/d032/d046/d080    BOPOMOFO LETTER J
<q4>     /d032/d032/d046/d081    BOPOMOFO LETTER Q
<x4>     /d032/d032/d046/d082    BOPOMOFO LETTER X
<zh>     /d032/d032/d046/d083    BOPOMOFO LETTER ZH
<ch>     /d032/d032/d046/d084    BOPOMOFO LETTER CH
<sh>     /d032/d032/d046/d085    BOPOMOFO LETTER SH
<r4>     /d032/d032/d046/d086    BOPOMOFO LETTER R
<z4>     /d032/d032/d046/d087    BOPOMOFO LETTER Z
<c4>     /d032/d032/d046/d088    BOPOMOFO LETTER C
<s4>     /d032/d032/d046/d089    BOPOMOFO LETTER S
<a4>     /d032/d032/d046/d090    BOPOMOFO LETTER A
<o4>     /d032/d032/d046/d091    BOPOMOFO LETTER O
<e4>     /d032/d032/d046/d092    BOPOMOFO LETTER E
<eh>     /d032/d032/d046/d093    BOPOMOFO LETTER EH
<ai>     /d032/d032/d046/d094    BOPOMOFO LETTER AI
<ei>     /d032/d032/d046/d095    BOPOMOFO LETTER EI
<au>     /d032/d032/d046/d096    BOPOMOFO LETTER AU
<ou>     /d032/d032/d046/d097    BOPOMOFO LETTER OU
<an>     /d032/d032/d046/d098    BOPOMOFO LETTER AN
<en>     /d032/d032/d046/d099    BOPOMOFO LETTER EN
<aN>     /d032/d032/d046/d100    BOPOMOFO LETTER ANG
<eN>     /d032/d032/d046/d101    BOPOMOFO LETTER ENG
<er>     /d032/d032/d046/d102    BOPOMOFO LETTER ER
<i4>     /d032/d032/d046/d103    BOPOMOFO LETTER I
<u4>     /d032/d032/d046/d104    BOPOMOFO LETTER U
<iu>     /d032/d032/d046/d105    BOPOMOFO LETTER IU
<A5>     /d032/d032/d047/d033    HIRAGANA LETTER SMALL A
<a5>     /d032/d032/d047/d034    HIRAGANA LETTER A
<I5>     /d032/d032/d047/d035    HIRAGANA LETTER SMALL I
<i5>     /d032/d032/d047/d036    HIRAGANA LETTER I
<U5>     /d032/d032/d047/d037    HIRAGANA LETTER SMALL U
<u5>     /d032/d032/d047/d038    HIRAGANA LETTER U
<E5>     /d032/d032/d047/d039    HIRAGANA LETTER SMALL E
<e5>     /d032/d032/d047/d040    HIRAGANA LETTER E
<O5>     /d032/d032/d047/d041    HIRAGANA LETTER SMALL O
<o5>     /d032/d032/d047/d042    HIRAGANA LETTER O
<ka>     /d032/d032/d047/d043    HIRAGANA LETTER KA
<ga>     /d032/d032/d047/d044    HIRAGANA LETTER GA
<ki>     /d032/d032/d047/d045    HIRAGANA LETTER KI
<gi>     /d032/d032/d047/d046    HIRAGANA LETTER GI
<ku>     /d032/d032/d047/d047    HIRAGANA LETTER KU
<gu>     /d032/d032/d047/d048    HIRAGANA LETTER GU
<ke>     /d032/d032/d047/d049    HIRAGANA LETTER KE
<ge>     /d032/d032/d047/d050    HIRAGANA LETTER GE
<ko>     /d032/d032/d047/d051    HIRAGANA LETTER KO
<go>     /d032/d032/d047/d052    HIRAGANA LETTER GO
<sa>     /d032/d032/d047/d053    HIRAGANA LETTER SA
<za>     /d032/d032/d047/d054    HIRAGANA LETTER ZA
<si>     /d032/d032/d047/d055    HIRAGANA LETTER SI
<zi>     /d032/d032/d047/d056    HIRAGANA LETTER ZI
<su>     /d032/d032/d047/d057    HIRAGANA LETTER SU
<zu>     /d032/d032/d047/d058    HIRAGANA LETTER ZU
<se>     /d032/d032/d047/d059    HIRAGANA LETTER SE
<ze>     /d032/d032/d047/d060    HIRAGANA LETTER ZE
<so>     /d032/d032/d047/d061    HIRAGANA LETTER SO
```

```
<zo>    /d032/d032/d047/d062    HIRAGANA LETTER ZO
<ta>    /d032/d032/d047/d063    HIRAGANA LETTER TA
<da>    /d032/d032/d047/d064    HIRAGANA LETTER DA
<ti>    /d032/d032/d047/d065    HIRAGANA LETTER TI
<di>    /d032/d032/d047/d066    HIRAGANA LETTER DI
<tU>    /d032/d032/d047/d067    HIRAGANA LETTER SMALL TU
<tu>    /d032/d032/d047/d068    HIRAGANA LETTER TU
<du>    /d032/d032/d047/d069    HIRAGANA LETTER DU
<te>    /d032/d032/d047/d070    HIRAGANA LETTER TE
<de>    /d032/d032/d047/d071    HIRAGANA LETTER DE
<to>    /d032/d032/d047/d072    HIRAGANA LETTER TO
<do>    /d032/d032/d047/d073    HIRAGANA LETTER DO
<na>    /d032/d032/d047/d074    HIRAGANA LETTER NA
<ni>    /d032/d032/d047/d075    HIRAGANA LETTER NI
<nu>    /d032/d032/d047/d076    HIRAGANA LETTER NU
<ne>    /d032/d032/d047/d077    HIRAGANA LETTER NE
<no>    /d032/d032/d047/d078    HIRAGANA LETTER NO
<ha>    /d032/d032/d047/d079    HIRAGANA LETTER HA
<ba>    /d032/d032/d047/d080    HIRAGANA LETTER BA
<pa>    /d032/d032/d047/d081    HIRAGANA LETTER PA
<hi>    /d032/d032/d047/d082    HIRAGANA LETTER HI
<bi>    /d032/d032/d047/d083    HIRAGANA LETTER BI
<pi>    /d032/d032/d047/d084    HIRAGANA LETTER PI
<hu>    /d032/d032/d047/d085    HIRAGANA LETTER HU
<bu>    /d032/d032/d047/d086    HIRAGANA LETTER BU
<pu>    /d032/d032/d047/d087    HIRAGANA LETTER PU
<he>    /d032/d032/d047/d088    HIRAGANA LETTER HE
<be>    /d032/d032/d047/d089    HIRAGANA LETTER BE
<pe>    /d032/d032/d047/d090    HIRAGANA LETTER PE
<ho>    /d032/d032/d047/d091    HIRAGANA LETTER HO
<bo>    /d032/d032/d047/d092    HIRAGANA LETTER BO
<po>    /d032/d032/d047/d093    HIRAGANA LETTER PO
<ma>    /d032/d032/d047/d094    HIRAGANA LETTER MA
<mi>    /d032/d032/d047/d095    HIRAGANA LETTER MI
<mu>    /d032/d032/d047/d096    HIRAGANA LETTER MU
<me>    /d032/d032/d047/d097    HIRAGANA LETTER ME
<mo>    /d032/d032/d047/d098    HIRAGANA LETTER MO
<yA>    /d032/d032/d047/d099    HIRAGANA LETTER SMALL YA
<ya>    /d032/d032/d047/d100    HIRAGANA LETTER YA
<yU>    /d032/d032/d047/d101    HIRAGANA LETTER SMALL YU
<yu>    /d032/d032/d047/d102    HIRAGANA LETTER YU
<yO>    /d032/d032/d047/d103    HIRAGANA LETTER SMALL YO
<yo>    /d032/d032/d047/d104    HIRAGANA LETTER YO                          1
<ra>    /d032/d032/d047/d105    HIRAGANA LETTER RA
<ri>    /d032/d032/d047/d106    HIRAGANA LETTER RI
<ru>    /d032/d032/d047/d107    HIRAGANA LETTER RU
<re>    /d032/d032/d047/d108    HIRAGANA LETTER RE
<ro>    /d032/d032/d047/d109    HIRAGANA LETTER RO
<wA>    /d032/d032/d047/d110    HIRAGANA LETTER SMALL WA
<wa>    /d032/d032/d047/d111    HIRAGANA LETTER WA
<wi>    /d032/d032/d047/d112    HIRAGANA LETTER WI
<we>    /d032/d032/d047/d113    HIRAGANA LETTER WE
<wo>    /d032/d032/d047/d114    HIRAGANA LETTER WO
<n5>    /d032/d032/d047/d115    HIRAGANA LETTER N
<"5>    /d032/d032/d047/d122    HIRAGANA-KATAKANA VOICED SOUND MARK
<05>    /d032/d032/d047/d123    HIRAGANA-KATAKANA SEMI-VOICED SOUND MARK
<*5>    /d032/d032/d047/d124    HIRAGANA ITERATION MARK
```

```
<+5>    /d032/d032/d047/d125    HIRAGANA VOICED ITERATION MARK
<a6>    /d032/d032/d047/d161    KATAKANA LETTER SMALL A
<A6>    /d032/d032/d047/d162    KATAKANA LETTER A
<i6>    /d032/d032/d047/d163    KATAKANA LETTER SMALL I
<I6>    /d032/d032/d047/d164    KATAKANA LETTER I
<u6>    /d032/d032/d047/d165    KATAKANA LETTER SMALL U
<U6>    /d032/d032/d047/d166    KATAKANA LETTER U
<e6>    /d032/d032/d047/d167    KATAKANA LETTER SMALL E
<E6>    /d032/d032/d047/d168    KATAKANA LETTER E
<o6>    /d032/d032/d047/d169    KATAKANA LETTER SMALL O
<O6>    /d032/d032/d047/d170    KATAKANA LETTER O
<Ka>    /d032/d032/d047/d171    KATAKANA LETTER KA
<Ga>    /d032/d032/d047/d172    KATAKANA LETTER GA
<Ki>    /d032/d032/d047/d173    KATAKANA LETTER KI
<Gi>    /d032/d032/d047/d174    KATAKANA LETTER GI
<Ku>    /d032/d032/d047/d175    KATAKANA LETTER KU
<Gu>    /d032/d032/d047/d176    KATAKANA LETTER GU
<Ke>    /d032/d032/d047/d177    KATAKANA LETTER KE
<Ge>    /d032/d032/d047/d178    KATAKANA LETTER GE
<Ko>    /d032/d032/d047/d179    KATAKANA LETTER KO
<Go>    /d032/d032/d047/d180    KATAKANA LETTER GO
<Sa>    /d032/d032/d047/d181    KATAKANA LETTER SA
<Za>    /d032/d032/d047/d182    KATAKANA LETTER ZA
<Si>    /d032/d032/d047/d183    KATAKANA LETTER SI
<Zi>    /d032/d032/d047/d184    KATAKANA LETTER ZI
<Su>    /d032/d032/d047/d185    KATAKANA LETTER SU
<Zu>    /d032/d032/d047/d186    KATAKANA LETTER ZU
<Se>    /d032/d032/d047/d187    KATAKANA LETTER SE
<Ze>    /d032/d032/d047/d188    KATAKANA LETTER ZE
<So>    /d032/d032/d047/d189    KATAKANA LETTER SO
<Zo>    /d032/d032/d047/d190    KATAKANA LETTER ZO
<Ta>    /d032/d032/d047/d191    KATAKANA LETTER TA
<Da>    /d032/d032/d047/d192    KATAKANA LETTER DA
<Ti>    /d032/d032/d047/d193    KATAKANA LETTER TI
<Di>    /d032/d032/d047/d194    KATAKANA LETTER DI
<TU>    /d032/d032/d047/d195    KATAKANA LETTER SMALL TU
<Tu>    /d032/d032/d047/d196    KATAKANA LETTER TU
<Du>    /d032/d032/d047/d197    KATAKANA LETTER DU
<Te>    /d032/d032/d047/d198    KATAKANA LETTER TE
<De>    /d032/d032/d047/d199    KATAKANA LETTER DE
<To>    /d032/d032/d047/d200    KATAKANA LETTER TO
<Do>    /d032/d032/d047/d201    KATAKANA LETTER DO
<Na>    /d032/d032/d047/d202    KATAKANA LETTER NA
<Ni>    /d032/d032/d047/d203    KATAKANA LETTER NI
<Nu>    /d032/d032/d047/d204    KATAKANA LETTER NU
<Ne>    /d032/d032/d047/d205    KATAKANA LETTER NE
<No>    /d032/d032/d047/d206    KATAKANA LETTER NO
<Ha>    /d032/d032/d047/d207    KATAKANA LETTER HA
<Ba>    /d032/d032/d047/d208    KATAKANA LETTER BA
<Pa>    /d032/d032/d047/d209    KATAKANA LETTER PA
<Hi>    /d032/d032/d047/d210    KATAKANA LETTER HI
<Bi>    /d032/d032/d047/d211    KATAKANA LETTER BI                1
<Pi>    /d032/d032/d047/d212    KATAKANA LETTER PI                1
<Hu>    /d032/d032/d047/d213    KATAKANA LETTER HU
<Bu>    /d032/d032/d047/d214    KATAKANA LETTER BU
<Pu>    /d032/d032/d047/d215    KATAKANA LETTER PU
<He>    /d032/d032/d047/d216    KATAKANA LETTER HE
```

```
<Be>    /d032/d032/d047/d217    KATAKANA LETTER BE
<Pe>    /d032/d032/d047/d218    KATAKANA LETTER PE
<Ho>    /d032/d032/d047/d219    KATAKANA LETTER HO
<Bo>    /d032/d032/d047/d220    KATAKANA LETTER BO
<Po>    /d032/d032/d047/d221    KATAKANA LETTER PO
<Ma>    /d032/d032/d047/d222    KATAKANA LETTER MA
<Mi>    /d032/d032/d047/d223    KATAKANA LETTER MI
<Mu>    /d032/d032/d047/d224    KATAKANA LETTER MU
<Me>    /d032/d032/d047/d225    KATAKANA LETTER ME
<Mo>    /d032/d032/d047/d226    KATAKANA LETTER MO
<YA>    /d032/d032/d047/d227    KATAKANA LETTER SMALL YA
<Ya>    /d032/d032/d047/d228    KATAKANA LETTER YA
<YU>    /d032/d032/d047/d229    KATAKANA LETTER SMALL YU
<Yu>    /d032/d032/d047/d230    KATAKANA LETTER YU
<YO>    /d032/d032/d047/d231    KATAKANA LETTER SMALL YO
<Yo>    /d032/d032/d047/d232    KATAKANA LETTER YO
<Ra>    /d032/d032/d047/d233    KATAKANA LETTER RA
<Ri>    /d032/d032/d047/d234    KATAKANA LETTER RI
<Ru>    /d032/d032/d047/d235    KATAKANA LETTER RU
<Re>    /d032/d032/d047/d236    KATAKANA LETTER RE
<Ro>    /d032/d032/d047/d237    KATAKANA LETTER RO
<WA>    /d032/d032/d047/d238    KATAKANA LETTER SMALL WA
<Wa>    /d032/d032/d047/d239    KATAKANA LETTER WA
<Wi>    /d032/d032/d047/d240    KATAKANA LETTER WI
<We>    /d032/d032/d047/d241    KATAKANA LETTER WE
<Wo>    /d032/d032/d047/d242    KATAKANA LETTER WO
<N6>    /d032/d032/d047/d243    KATAKANA LETTER N
<Vu>    /d032/d032/d047/d244    KATAKANA LETTER VU
<KA>    /d032/d032/d047/d245    KATAKANA LETTER SMALL KA
<KE>    /d032/d032/d047/d246    KATAKANA LETTER SMALL KE
<-6>    /d032/d032/d047/d252    HIRAGANA-KATAKANA PROLONGED SOUND MARK
<*6>    /d032/d032/d047/d253    KATAKANA ITERATION MARK
<+6>    /d032/d032/d047/d254    KATAKANA VOICED ITERATION MARK
<ff>    /d032/d032/d060/d040    LATIN SMALL LIGATURE FF
<fi>    /d032/d032/d060/d041    LATIN SMALL LIGATURE FI
<fl>    /d032/d032/d060/d042    LATIN SMALL LIGATURE FL
<ft>    /d032/d032/d060/d045    LATIN SMALL LIGATURE FT
<st>    /d032/d032/d060/d046    LATIN SMALL LIGATURE ST
<Iu>    /d032/d032/d060/d048    INTEGRAL SIGN UPPER PART
<Il>    /d032/d032/d060/d049    INTEGRAL SIGN LOWER PART
<NU>    /d000/d128/d128/d128    NULL (NUL)                             1
<SH>    /d001/d128/d128/d128    START OF HEADING (SOH)                 1
<SX>    /d002/d128/d128/d128    START OF TEXT (STX)                    1
<EX>    /d003/d128/d128/d128    END OF TEXT (ETX)                      1
<ET>    /d004/d128/d128/d128    END OF TRANSMISSION (EOT)              1
<EQ>    /d005/d128/d128/d128    ENQUIRY (ENQ)                          1
<AK>    /d006/d128/d128/d128    ACKNOWLEDGE (ACK)                      1
<BL>    /d007/d128/d128/d128    BELL (BEL)                             1
<BS>    /d008/d128/d128/d128    BACKSPACE (BS)                         1
<HT>    /d009/d128/d128/d128    CHARACTER TABULATION (HT)              1
<LF>    /d010/d128/d128/d128    LINE FEED (LF)                         1
<VT>    /d011/d128/d128/d128    LINE TABULATION (VT)                   1
<FF>    /d012/d128/d128/d128    FORM FEED (FF)                         1
<CR>    /d013/d128/d128/d128    CARRIAGE RETURN (CR)                   1
<SO>    /d014/d128/d128/d128    SHIFT OUT (SO)                         1
<SI>    /d015/d128/d128/d128    SHIFT IN (SI)                          1
<DL>    /d016/d128/d128/d128    DATALINK ESCAPE (DLE)                  1
```

```
<D1>     /d017/d128/d128/d128    DEVICE CONTROL ONE (DC1)                             1
<D2>     /d018/d128/d128/d128    DEVICE CONTROL TWO (DC2)                             1
<D3>     /d019/d128/d128/d128    DEVICE CONTROL THREE (DC3)                           1
<D4>     /d020/d128/d128/d128    DEVICE CONTROL FOUR (DC4)                            1
<NK>     /d021/d128/d128/d128    NEGATIVE ACKNOWLEDGE (NAK)                           1
<SY>     /d022/d128/d128/d128    SYNCHRONOUS IDLE (SYN)                               1
<EB>     /d023/d128/d128/d128    END OF TRANSMISSION BLOCK (ETB)                      1
<CN>     /d024/d128/d128/d128    CANCEL (CAN)                                         1
<EM>     /d025/d128/d128/d128    END OF MEDIUM (EM)                                   1
<SB>     /d026/d128/d128/d128    SUBSTITUTE (SUB)                                     1
<EC>     /d027/d128/d128/d128    ESCAPE (ESC)                                         1
<FS>     /d028/d128/d128/d128    FILE SEPARATOR (IS4)                                 1
<GS>     /d029/d128/d128/d128    GROUP SEPARATOR (IS3)                                1
<RS>     /d030/d128/d128/d128    RECORD SEPARATOR (IS2)                               1
<US>     /d031/d128/d128/d128    UNIT SEPARATOR (IS1)                                 1
<DT>     /d127/d128/d128/d128    DELETE (DEL)                                         1
<PA>     /d128/d128/d128/d128    PADDING CHARACTER (PAD)                              1
<HO>     /d129/d128/d128/d128    HIGH OCTET PRESET (HOP)                              1
<BH>     /d130/d128/d128/d128    BREAK PERMITTED HERE (BPH)                           1
<NH>     /d131/d128/d128/d128    NO BREAK HERE (NBH)                                  1
<IN>     /d132/d128/d128/d128    INDEX (IND)                                          1
<NL>     /d133/d128/d128/d128    NEXT LINE (NEL)                                      1
<SA>     /d134/d128/d128/d128    START OF SELECTED AREA (SSA)                         1
<ES>     /d135/d128/d128/d128    END OF SELECTED AREA (ESA)                           1
<HS>     /d136/d128/d128/d128    CHARACTER TABULATION SET (HTS)                       1
<HJ>     /d137/d128/d128/d128    CHARACTER TABULATION WITH JUSTIFICATION (HTJ)        1
<VS>     /d138/d128/d128/d128    LINE TABULATION SET (VTS)                            1
<PD>     /d139/d128/d128/d128    PARTIAL LINE FORWARD (PLD)                           1
<PU>     /d140/d128/d128/d128    PARTIAL LINE BACKWARD (PLU)                          1
<RI>     /d141/d128/d128/d128    REVERSE LINE FEED (RI)                               1
<S2>     /d142/d128/d128/d128    SINGLE-SHIFT TWO (SS2)                               1
<S3>     /d143/d128/d128/d128    SINGLE-SHIFT THREE (SS3)                             1
<DC>     /d144/d128/d128/d128    DEVICE CONTROL STRING (DCS)                          1
<P1>     /d145/d128/d128/d128    PRIVATE USE ONE (PU1)                                1
<P2>     /d146/d128/d128/d128    PRIVATE USE TWO (PU2)                                1
<TS>     /d147/d128/d128/d128    SET TRANSMIT STATE (STS)                             1
<CC>     /d148/d128/d128/d128    CANCEL CHARACTER (CCH)                               1
<MW>     /d149/d128/d128/d128    MESSAGE WAITING (MW)                                 1
<SG>     /d150/d128/d128/d128    START OF GUARDED AREA (SPA)                          1
<EG>     /d151/d128/d128/d128    END OF GUARDED AREA (EPA)                            1
<SS>     /d152/d128/d128/d128    START OF STRING (SOS)                                1
<GC>     /d153/d128/d128/d128    SINGLE GRAPHIC CHARACTER INTRODUCER (SGCI)           1
<SC>     /d154/d128/d128/d128    SINGLE CHARACTER INTRODUCER (SCI)                    1
<CI>     /d155/d128/d128/d128    CONTROL SEQUENCE INTRODUCER (CSI)                    1
<ST>     /d156/d128/d128/d128    STRING TERMINATOR (ST)                               1
<OC>     /d157/d128/d128/d128    OPERATING SYSTEM COMMAND (OSC)                       1
<PM>     /d158/d128/d128/d128    PRIVACY MESSAGE (PM)                                 1
<AC>     /d159/d128/d128/d128    APPLICATION PROGRAM COMMAND (APC)                    1
<__>     /d032/d032/d052/d032    indicates unfinished
<"!>     /d032/d032/d052/d033    NON-SPACING GRAVE ACCENT (ISO IR 70 193)
<"'>     /d032/d032/d052/d034    NON-SPACING ACUTE ACCENT (ISO IR 70 194)
<"/>>    /d032/d032/d052/d035    NON-SPACING CIRCUMFLEX ACCENT (ISO IR 70 195)
<"?>     /d032/d032/d052/d036    NON-SPACING TILDE (ISO IR 70 196)
<"->     /d032/d032/d052/d037    NON-SPACING MACRON (ISO IR 70 197)
<"(>     /d032/d032/d052/d038    NON-SPACING BREVE (ISO IR 70 198)
<".>     /d032/d032/d052/d039    NON-SPACING DOT ABOVE (ISO IR 70 199)
<":>     /d032/d032/d052/d040    NON-SPACING DIAERESIS (ISO IR 70 200)
```

```
<"//>   /d032/d032/d052/d041    NON-SPACING SOLIDUS (ISO IR 99 201)
<"0>    /d032/d032/d052/d042    NON-SPACING RING ABOVE (ISO IR 70 202)
<",>    /d032/d032/d052/d043    NON-SPACING CEDILLA (ISO IR 70 203)
<"_>    /d032/d032/d052/d044    NON-SPACING UNDERLINE (ISO IR 99 216)
<"">    /d032/d032/d052/d045    NON-SPACING DOUBLE ACCUTE ACCENT (ISO IR 70 205)
<"<>    /d032/d032/d052/d046    NON-SPACING CARON (ISO IR 70 207)
<";>    /d032/d032/d052/d047    NON-SPACING OGONEK (ISO IR 53 208)
<"=>    /d032/d032/d052/d048    NON-SPACING DOUBLE UNDERLINE (ISO IR 53 217)
<"1>    /d032/d032/d052/d049    NON-SPACING DIAERESIS WITH ACCENT
#                               (ISO IR 70 192)
<"2>    /d032/d032/d052/d050    NON-SPACING UMLAUT (ISO 5426 201)
<Fd>    /d032/d032/d052/d051    FILLED FORWARD DIAGONAL
#                               (ANSI X3.110-1983 218)
<Bd>    /d032/d032/d052/d052    FILLED BACKWARD DIAGONAL
#                               (ANSI X3.110-1983 219)
<Fl>    /d032/d032/d052/d053    Dutch guilder sign (IBM CP 437 159)
<Li>    /d032/d032/d052/d054    Italian Lira sign (HP ROMAN 8 175)
<//f>   /d032/d032/d052/d055    VULGAR FRACTION BAR (MacIntosh 218)
<0s>    /d032/d032/d052/d056    SUBSCRIPT ZERO (ISO IR 50 096)
<1s>    /d032/d032/d052/d057    SUBSCRIPT ONE (ISO IR 50 097)
<2s>    /d032/d032/d052/d058    SUBSCRIPT TWO (ISO IR 50 098)
<3s>    /d032/d032/d052/d059    SUBSCRIPT THREE (ISO IR 50 099)
<4s>    /d032/d032/d052/d060    SUBSCRIPT FOUR (ISO IR 50 100)
<5s>    /d032/d032/d052/d061    SUBSCRIPT FIVE (ISO IR 50 101)
<6s>    /d032/d032/d052/d062    SUBSCRIPT SIX (ISO IR 50 102)
<7s>    /d032/d032/d052/d063    SUBSCRIPT SEVEN (ISO IR 50 103)
<8s>    /d032/d032/d052/d064    SUBSCRIPT EIGHT (ISO IR 50 104)
<9s>    /d032/d032/d052/d065    SUBSCRIPT NINE (ISO IR 50 105)
<0S>    /d032/d032/d052/d066    SUPERSCRIPT ZERO (ISO IR 50 112)
<4S>    /d032/d032/d052/d067    SUPERSCRIPT FOUR (ISO IR 50 116)
<5S>    /d032/d032/d052/d068    SUPERSCRIPT FIVE (ISO IR 50 117)
<6S>    /d032/d032/d052/d069    SUPERSCRIPT SIX (ISO IR 50 118)
<7S>    /d032/d032/d052/d070    SUPERSCRIPT SEVEN (ISO IR 50 119)
<8S>    /d032/d032/d052/d071    SUPERSCRIPT EIGHT (ISO IR 50 120)
<9S>    /d032/d032/d052/d072    SUPERSCRIPT NINE (ISO IR 50 121)
<+S>    /d032/d032/d052/d073    SUPERSCRIPT PLUS (ISO IR 50 106)
<-S>    /d032/d032/d052/d074    SUPERSCRIPT MINUS (ISO IR 50 107)
<1h>    /d032/d032/d052/d075    ABSTRACT SYMBOL H ONE (HOOK)
#                               (JIS C 6229-1984 060)
<2h>    /d032/d032/d052/d076    ABSTRACT SYMBOL H TWO (FORK)
#                               (JIS C 6229-1984 093)
<3h>    /d032/d032/d052/d077    ABSTRACT SYMBOL H THREE (CHAIR)
#                               (JIS C 6229-1984 062)
<4h>    /d032/d032/d052/d078    ABSTRACT SYMBOL H FOUR (LONG VERTICAL MARK)
#                               (JIS C 6229-1984 125)
<1j>    /d032/d032/d052/d079    SYMBOL ONE (ISO 2033-1983 058)
<2j>    /d032/d032/d052/d080    SYMBOL TWO (ISO 2033-1983 059)
<3j>    /d032/d032/d052/d081    SYMBOL THREE (ISO 2033-1983 060)
<4j>    /d032/d032/d052/d082    SYMBOL FOUR (ISO 2033-1983 061)
<UA>    /d032/d032/d052/d083    Unit space A (ISO IR 8-1 064)
<UB>    /d032/d032/d052/d084    Unit space B (ISO IR 8-1 096)
<yf>    /d032/d032/d052/d085    ARABIC LETTER YEH FINAL (CODAR U 090)
<yr>    /d032/d032/d052/d086    OLD NORSE YR (DIN 31624 251)
<.6>    /d032/d032/d052/d087    KATAKANA FULL STOP (JIS C 6220 033)
<<6>    /d032/d032/d052/d088    KATAKANA OPENING BRACKET (JIS C 6220 034)
</>6>   /d032/d032/d052/d089    KATAKANA CLOSING BRACKET (JIS C 6220 035)
<,6>    /d032/d032/d052/d090    KATAKANA COMMA (JIS C 6220 036)
```

```
<&6>    /d032/d032/d052/d091    KATAKANA CONJUNCTION SYMBOL (JIS C 6220 037)
<(S>    /d032/d032/d052/d092    LEFT PARENTHESIS SUPERSCRIPT
#                               (CSA Z243.4-1985-gr 168)
<)S>    /d032/d032/d052/d093    RIGHT PARENTHESIS SUPERSCRIPT
#                               (CSA Z243.4-1985-gr 169)
END CHARMAP
```

F.5.2 ISO_8859-1 Charmap

```
<escape_char> /
<mb_cur_max> 1
CHARMAP
<NUL>                   /d000 NULL (NUL)                                1
<SOH>                   /d001 START OF HEADING (SOH)
<STX>                   /d002 START OF TEXT (STX)
<ETX>                   /d003 END OF TEXT (ETX)
<EOT>                   /d004 END OF TRANSMISSION (EOT)
<ENQ>                   /d005 ENQUIRY (ENQ)
<ACK>                   /d006 ACKNOWLEDGE (ACK)
<alert>                 /d007 BELL (BEL)
<BEL>                   /d007 BELL (BEL)
<backspace>             /d008 BACKSPACE (BS)
<tab>                   /d009 CHARACTER TABULATION (HT)
<newline>               /d010 LINE FEED (LF)
<vertical-tab>          /d011 LINE TABULATION (VT)
<form-feed>             /d012 FORM FEED (FF)
<carriage-return>       /d013 CARRIAGE RETURN (CR)
<DLE>                   /d016 DATALINK ESCAPE (DLE)
<DC1>                   /d017 DEVICE CONTROL ONE (DC1)
<DC2>                   /d018 DEVICE CONTROL TWO (DC2)
<DC3>                   /d019 DEVICE CONTROL THREE (DC3)
<DC4>                   /d020 DEVICE CONTROL FOUR (DC4)
<NAK>                   /d021 NEGATIVE ACKNOWLEDGE (NAK)
<SYN>                   /d022 SYNCHRONOUS IDLE (SYN)
<ETB>                   /d023 END OF TRANSMISSION BLOCK (ETB)
<CAN>                   /d024 CANCEL (CAN)
<SUB>                   /d026 SUBSTITUTE (SUB)
<ESC>                   /d027 ESCAPE (ESC)
<IS4>                   /d028 FILE SEPARATOR (IS4)
<IS3>                   /d029 GROUP SEPARATOR (IS3)
<intro>                 /d029 GROUP SEPARATOR (IS3)
<IS2>                   /d030 RECORD SEPARATOR (IS2)
<IS1>                   /d031 UNIT SEPARATOR (IS1)
<DEL>                   /d127 DELETE (DEL)                              1
<space>                 /d032   SPACE
<exclamation-mark>      /d033   EXCLAMATION MARK
<quotation-mark>        /d034   QUOTATION MARK
<number-sign>           /d035   NUMBER SIGN
<dollar-sign>           /d036   DOLLAR SIGN
<percent-sign>          /d037   PERCENT SIGN
<ampersand>             /d038   AMPERSAND
<apostrophe>            /d039   APOSTROPHE
<left-parenthesis>      /d040   LEFT PARENTHESIS
<right-parenthesis>     /d041   RIGHT PARENTHESIS
<asterisk>              /d042   ASTERISK
<plus-sign>             /d043   PLUS SIGN
```

```
<comma>                 /d044   COMMA
<hyphen>                /d045   HYPHEN-MINUS
<hyphen-minus>          /d045   HYPHEN-MINUS
<period>                /d046   FULL STOP
<full-stop>             /d046   FULL STOP
<slash>                 /d047   SOLIDUS
<solidus>               /d047   SOLIDUS
<zero>                  /d048   DIGIT ZERO
<one>                   /d049   DIGIT ONE
<two>                   /d050   DIGIT TWO
<three>                 /d051   DIGIT THREE
<four>                  /d052   DIGIT FOUR
<five>                  /d053   DIGIT FIVE
<six>                   /d054   DIGIT SIX
<seven>                 /d055   DIGIT SEVEN
<eight>                 /d056   DIGIT EIGHT
<nine>                  /d057   DIGIT NINE
<colon>                 /d058   COLON
<semicolon>             /d059   SEMICOLON
<less-than-sign>        /d060   LESS-THAN SIGN
<equals-sign>           /d061   EQUALS SIGN
<greater-than-sign>     /d062   GREATER-THAN SIGN
<question-mark>         /d063   QUESTION MARK
<commercial-at>         /d064   COMMERCIAL AT
<left-square-bracket>   /d091   LEFT SQUARE BRACKET
<reverse-solidus>       /d092   REVERSE SOLIDUS
<backslash>             /d092   REVERSE SOLIDUS
<right-square-bracket>  /d093   RIGHT SQUARE BRACKET
<circumflex-accent>     /d094   CIRCUMFLEX ACCENT
<low-line>              /d095   LOW LINE
<underscore>            /d095   LOW LINE
<grave-accent>          /d096   GRAVE ACCENT
<left-curly-bracket>    /d123   LEFT CURLY BRACKET
<vertical-line>         /d124   VERTICAL LINE
<right-curly-bracket>   /d125   RIGHT CURLY BRACKET
<tilde>                 /d126   TILDE
<SP>    /d032   SPACE
<!>     /d033   EXCLAMATION MARK
<">     /d034   QUOTATION MARK
<Nb>    /d035   NUMBER SIGN
<DO>    /d036   DOLLAR SIGN
<%>     /d037   PERCENT SIGN
<&>     /d038   AMPERSAND
<'>     /d039   APOSTROPHE
<(>     /d040   LEFT PARENTHESIS
<)>     /d041   RIGHT PARENTHESIS
<*>     /d042   ASTERISK
<+>     /d043   PLUS SIGN
<,>     /d044   COMMA
<->     /d045   HYPHEN-MINUS
<.>     /d046   FULL STOP
<//>    /d047   SOLIDUS
<0>     /d048   DIGIT ZERO
<1>     /d049   DIGIT ONE
<2>     /d050   DIGIT TWO
<3>     /d051   DIGIT THREE
<4>     /d052   DIGIT FOUR
```

```
<5>     /d053   DIGIT FIVE
<6>     /d054   DIGIT SIX
<7>     /d055   DIGIT SEVEN
<8>     /d056   DIGIT EIGHT
<9>     /d057   DIGIT NINE
<:>     /d058   COLON
<;>     /d059   SEMICOLON
<<>     /d060   LESS-THAN SIGN
<=>     /d061   EQUALS SIGN
</>>    /d062   GREATER-THAN SIGN
<?>     /d063   QUESTION MARK
<At>    /d064   COMMERCIAL AT
<A>     /d065   LATIN CAPITAL LETTER A
<B>     /d066   LATIN CAPITAL LETTER B
<C>     /d067   LATIN CAPITAL LETTER C
<D>     /d068   LATIN CAPITAL LETTER D
<E>     /d069   LATIN CAPITAL LETTER E
<F>     /d070   LATIN CAPITAL LETTER F
<G>     /d071   LATIN CAPITAL LETTER G
<H>     /d072   LATIN CAPITAL LETTER H
<I>     /d073   LATIN CAPITAL LETTER I
<J>     /d074   LATIN CAPITAL LETTER J
<K>     /d075   LATIN CAPITAL LETTER K
<L>     /d076   LATIN CAPITAL LETTER L
<M>     /d077   LATIN CAPITAL LETTER M
<N>     /d078   LATIN CAPITAL LETTER N
<O>     /d079   LATIN CAPITAL LETTER O
<P>     /d080   LATIN CAPITAL LETTER P
<Q>     /d081   LATIN CAPITAL LETTER Q
<R>     /d082   LATIN CAPITAL LETTER R
<S>     /d083   LATIN CAPITAL LETTER S
<T>     /d084   LATIN CAPITAL LETTER T
<U>     /d085   LATIN CAPITAL LETTER U
<V>     /d086   LATIN CAPITAL LETTER V
<W>     /d087   LATIN CAPITAL LETTER W
<X>     /d088   LATIN CAPITAL LETTER X
<Y>     /d089   LATIN CAPITAL LETTER Y
<Z>     /d090   LATIN CAPITAL LETTER Z
<<(>    /d091   LEFT SQUARE BRACKET
<////>  /d092   REVERSE SOLIDUS
<)/>>   /d093   RIGHT SQUARE BRACKET
<'/>>   /d094   CIRCUMFLEX ACCENT
<_>     /d095   LOW LINE
<'!>    /d096   GRAVE ACCENT
<a>     /d097   LATIN SMALL LETTER A
<b>     /d098   LATIN SMALL LETTER B
<c>     /d099   LATIN SMALL LETTER C
<d>     /d100   LATIN SMALL LETTER D
<e>     /d101   LATIN SMALL LETTER E
<f>     /d102   LATIN SMALL LETTER F
<g>     /d103   LATIN SMALL LETTER G
<h>     /d104   LATIN SMALL LETTER H
<i>     /d105   LATIN SMALL LETTER I
<j>     /d106   LATIN SMALL LETTER J
<k>     /d107   LATIN SMALL LETTER K
<l>     /d108   LATIN SMALL LETTER L
<m>     /d109   LATIN SMALL LETTER M
```

```
<n>      /d110   LATIN SMALL LETTER N
<o>      /d111   LATIN SMALL LETTER O
<p>      /d112   LATIN SMALL LETTER P
<q>      /d113   LATIN SMALL LETTER Q
<r>      /d114   LATIN SMALL LETTER R
<s>      /d115   LATIN SMALL LETTER S
<t>      /d116   LATIN SMALL LETTER T
<u>      /d117   LATIN SMALL LETTER U
<v>      /d118   LATIN SMALL LETTER V
<w>      /d119   LATIN SMALL LETTER W
<x>      /d120   LATIN SMALL LETTER X
<y>      /d121   LATIN SMALL LETTER Y
<z>      /d122   LATIN SMALL LETTER Z
<(!>     /d123   LEFT CURLY BRACKET
<!!>     /d124   VERTICAL LINE
<!)>     /d125   RIGHT CURLY BRACKET
<'?>     /d126   TILDE
<NS>     /d160   NO-BREAK SPACE
<!I>     /d161   INVERTED EXCLAMATION MARK
<Ct>     /d162   CENT SIGN
<Pd>     /d163   POUND SIGN
<Cu>     /d164   CURRENCY SIGN
<Ye>     /d165   YEN SIGN
<BB>     /d166   BROKEN BAR
<SE>     /d167   SECTION SIGN
<':>     /d168   DIAERESIS
<Co>     /d169   COPYRIGHT SIGN
<-a>     /d170   FEMININE ORDINAL INDICATOR
<<<>     /d171   LEFT POINTING DOUBLE ANGLE QUOTATION MARK
<NO>     /d172   NOT SIGN
<-->     /d173   SOFT HYPHEN
<Rg>     /d174   REGISTERED SIGN
<'->     /d175   MACRON
<DG>     /d176   DEGREE SIGN
<+->     /d177   PLUS-MINUS SIGN
<2S>     /d178   SUPERSCRIPT TWO
<3S>     /d179   SUPERSCRIPT THREE
<''>     /d180   ACUTE ACCENT
<My>     /d181   MICRO SIGN
<PI>     /d182   PILCROW SIGN
<.M>     /d183   MIDDLE DOT
<',>     /d184   CEDILLA
<1S>     /d185   SUPERSCRIPT ONE
<-o>     /d186   MASCULINE ORDINAL INDICATOR
</>>>>   /d187   RIGHT POINTING DOUBLE ANGLE QUOTATION MARK          1
<14>     /d188   VULGAR FRACTION ONE QUARTER
<12>     /d189   VULGAR FRACTION ONE HALF
<34>     /d190   VULGAR FRACTION THREE QUARTERS
<?I>     /d191   INVERTED QUESTION MARK
<A!>     /d192   LATIN CAPITAL LETTER A WITH GRAVE
<A'>     /d193   LATIN CAPITAL LETTER A WITH ACUTE
<A/>>    /d194   LATIN CAPITAL LETTER A WITH CIRCUMFLEX
<A?>     /d195   LATIN CAPITAL LETTER A WITH TILDE
<A:>     /d196   LATIN CAPITAL LETTER A WITH DIAERESIS
<AA>     /d197   LATIN CAPITAL LETTER A WITH RING ABOVE
<AE>     /d198   LATIN CAPITAL LETTER AE
<C,>     /d199   LATIN CAPITAL LETTER C WITH CEDILLA
```

```
<E!>    /d200   LATIN CAPITAL LETTER E WITH GRAVE
<E'>    /d201   LATIN CAPITAL LETTER E WITH ACUTE
<E/>>   /d202   LATIN CAPITAL LETTER E WITH CIRCUMFLEX
<E:>    /d203   LATIN CAPITAL LETTER E WITH DIAERESIS
<I!>    /d204   LATIN CAPITAL LETTER I WITH GRAVE
<I'>    /d205   LATIN CAPITAL LETTER I WITH ACUTE
<I/>>   /d206   LATIN CAPITAL LETTER I WITH CIRCUMFLEX
<I:>    /d207   LATIN CAPITAL LETTER I WITH DIAERESIS
<D->    /d208   LATIN CAPITAL LETTER ETH (Icelandic)
<N?>    /d209   LATIN CAPITAL LETTER N WITH TILDE
<O!>    /d210   LATIN CAPITAL LETTER O WITH GRAVE
<O'>    /d211   LATIN CAPITAL LETTER O WITH ACUTE
<O/>>   /d212   LATIN CAPITAL LETTER O WITH CIRCUMFLEX
<O?>    /d213   LATIN CAPITAL LETTER O WITH TILDE
<O:>    /d214   LATIN CAPITAL LETTER O WITH DIAERESIS
<*X>    /d215   MULTIPLICATION SIGN
<O//>   /d216   LATIN CAPITAL LETTER O WITH STROKE
<U!>    /d217   LATIN CAPITAL LETTER U WITH GRAVE
<U'>    /d218   LATIN CAPITAL LETTER U WITH ACUTE
<U/>>   /d219   LATIN CAPITAL LETTER U WITH CIRCUMFLEX
<U:>    /d220   LATIN CAPITAL LETTER U WITH DIAERESIS
<Y'>    /d221   LATIN CAPITAL LETTER Y WITH ACUTE
<TH>    /d222   LATIN CAPITAL LETTER THORN (Icelandic)
<ss>    /d223   LATIN SMALL LETTER SHARP S (German)
<a!>    /d224   LATIN SMALL LETTER A WITH GRAVE
<a'>    /d225   LATIN SMALL LETTER A WITH ACUTE
<a/>>   /d226   LATIN SMALL LETTER A WITH CIRCUMFLEX
<a?>    /d227   LATIN SMALL LETTER A WITH TILDE
<a:>    /d228   LATIN SMALL LETTER A WITH DIAERESIS
<aa>    /d229   LATIN SMALL LETTER A WITH RING ABOVE
<ae>    /d230   LATIN SMALL LETTER AE
<c,>    /d231   LATIN SMALL LETTER C WITH CEDILLA
<e!>    /d232   LATIN SMALL LETTER E WITH GRAVE
<e'>    /d233   LATIN SMALL LETTER E WITH ACUTE
<e/>>   /d234   LATIN SMALL LETTER E WITH CIRCUMFLEX
<e:>    /d235   LATIN SMALL LETTER E WITH DIAERESIS
<i!>    /d236   LATIN SMALL LETTER I WITH GRAVE
<i'>    /d237   LATIN SMALL LETTER I WITH ACUTE
<i/>>   /d238   LATIN SMALL LETTER I WITH CIRCUMFLEX
<i:>    /d239   LATIN SMALL LETTER I WITH DIAERESIS
<d->    /d240   LATIN SMALL LETTER ETH (Icelandic)
<n?>    /d241   LATIN SMALL LETTER N WITH TILDE
<o!>    /d242   LATIN SMALL LETTER O WITH GRAVE
<o'>    /d243   LATIN SMALL LETTER O WITH ACUTE
<o/>>   /d244   LATIN SMALL LETTER O WITH CIRCUMFLEX
<o?>    /d245   LATIN SMALL LETTER O WITH TILDE
<o:>    /d246   LATIN SMALL LETTER O WITH DIAERESIS
<-:>    /d247   DIVISION SIGN
<o//>   /d248   LATIN SMALL LETTER O WITH STROKE
<u!>    /d249   LATIN SMALL LETTER U WITH GRAVE
<u'>    /d250   LATIN SMALL LETTER U WITH ACUTE
<u/>>   /d251   LATIN SMALL LETTER U WITH CIRCUMFLEX
<u:>    /d252   LATIN SMALL LETTER U WITH DIAERESIS
<y'>    /d253   LATIN SMALL LETTER Y WITH ACUTE
<th>    /d254   LATIN SMALL LETTER THORN (Icelandic)
<y:>    /d255   LATIN SMALL LETTER Y WITH DIAERESIS
<NU>    /d000   NULL (NUL)
```

```
<SH>    /d001   START OF HEADING (SOH)
<SX>    /d002   START OF TEXT (STX)
<EX>    /d003   END OF TEXT (ETX)
<ET>    /d004   END OF TRANSMISSION (EOT)
<EQ>    /d005   ENQUIRY (ENQ)
<AK>    /d006   ACKNOWLEDGE (ACK)
<BL>    /d007   BELL (BEL)
<BS>    /d008   BACKSPACE (BS)
<HT>    /d009   CHARACTER TABULATION (HT)
<LF>    /d010   LINE FEED (LF)
<VT>    /d011   LINE TABULATION (VT)
<FF>    /d012   FORM FEED (FF)
<CR>    /d013   CARRIAGE RETURN (CR)
<SO>    /d014   SHIFT OUT (SO)
<SI>    /d015   SHIFT IN (SI)
<DL>    /d016   DATALINK ESCAPE (DLE)
<D1>    /d017   DEVICE CONTROL ONE (DC1)
<D2>    /d018   DEVICE CONTROL TWO (DC2)
<D3>    /d019   DEVICE CONTROL THREE (DC3)
<D4>    /d020   DEVICE CONTROL FOUR (DC4)
<NK>    /d021   NEGATIVE ACKNOWLEDGE (NAK)
<SY>    /d022   SYNCHRONOUS IDLE (SYN)
<EB>    /d023   END OF TRANSMISSION BLOCK (ETB)
<CN>    /d024   CANCEL (CAN)
<EM>    /d025   END OF MEDIUM (EM)
<SB>    /d026   SUBSTITUTE (SUB)
<EC>    /d027   ESCAPE (ESC)
<FS>    /d028   FILE SEPARATOR (IS4)
<GS>    /d029   GROUP SEPARATOR (IS3)
<RS>    /d030   RECORD SEPARATOR (IS2)
<US>    /d031   UNIT SEPARATOR (IS1)
<DT>    /d127   DELETE (DEL)
<PA>    /d128   PADDING CHARACTER (PAD)
<HO>    /d129   HIGH OCTET PRESET (HOP)
<BH>    /d130   BREAK PERMITTED HERE (BPH)
<NH>    /d131   NO BREAK HERE (NBH)
<IN>    /d132   INDEX (IND)
<NL>    /d133   NEXT LINE (NEL)
<SA>    /d134   START OF SELECTED AREA (SSA)
<ES>    /d135   END OF SELECTED AREA (ESA)
<HS>    /d136   CHARACTER TABULATION SET (HTS)
<HJ>    /d137   CHARACTER TABULATION WITH JUSTIFICATION (HTJ)
<VS>    /d138   LINE TABULATION SET (VTS)
<PD>    /d139   PARTIAL LINE FORWARD (PLD)
<PU>    /d140   PARTIAL LINE BACKWARD (PLU)
<RI>    /d141   REVERSE LINE FEED (RI)
<S2>    /d142   SINGLE-SHIFT TWO (SS2)
<S3>    /d143   SINGLE-SHIFT THREE (SS3)
<DC>    /d144   DEVICE CONTROL STRING (DCS)
<P1>    /d145   PRIVATE USE ONE (PU1)
<P2>    /d146   PRIVATE USE TWO (PU2)
<TS>    /d147   SET TRANSMIT STATE (STS)
<CC>    /d148   CANCEL CHARACTER (CCH)
<MW>    /d149   MESSAGE WAITING (MW)
<SG>    /d150   START OF GUARDED AREA (SPA)
<EG>    /d151   END OF GUARDED AREA (EPA)
<SS>    /d152   START OF STRING (SOS)
```

```
<GC>    /d153   SINGLE GRAPHIC CHARACTER INTRODUCER (SGCI)
<SC>    /d154   SINGLE CHARACTER INTRODUCER (SCI)
<CI>    /d155   CONTROL SEQUENCE INTRODUCER (CSI)
<ST>    /d156   STRING TERMINATOR (ST)
<OC>    /d157   OPERATING SYSTEM COMMAND (OSC)
<PM>    /d158   PRIVACY MESSAGE (PM)
<AC>    /d159   APPLICATION PROGRAM COMMAND (APC)
END CHARMAP
```

Annex G
(inform ative)

Balloting Instructions

This annex will not appear in the final standard. It is included in the draft to provide instructions for balloting that cannot be separated easily from the main document, as a cover letter might.

If you have received a copy of this draft before October 1991 it is important that you read this annex, whether you are an official member of the P1003.2 Balloting Group or not; comments on this draft are welcomed from all interested technical experts. **Your ballot is due to the IEEE office by 21 October 1991. This is not the date to postmark it—it is the date of receipt.**

Summary of Draft 11.2 Instructions

This is the fifth ‘‘recirculation draft’’ of P1003.2. The recirculation procedure is described in this annex. For this recirculation, we are accepting objections against any normative changes that occurred from Draft 11.1 to Draft 11.2 and the contents of the Unresolved Objections List, provided as a separate document from the draft.

This is the first ballot in which the draft is available for online review; see the Editor’s Notes for details on accessing this information.

Send your ballot and/or comments to:

IEEE Standards Office
Computer Society Secretariat
ATTN: P1003.2 Ballot (Anna Kaczmarek)
P.O. Box 1331
445 Hoes Lane
Piscataway, NJ 08855-1331

It would also be very helpful if you sent us your ballot in machine-readable form. Your official ballot must be returned via mail to the IEEE office; if we receive only the e-mail or diskette version, that version will not count as an official document. However, the online version would be a great help to ballot resolution. We can accept e-mail to the following address:

hlj@Posix.COM or **uunet!posix!hlj**

or IBM PC 3.5-inch/720K diskette (plain file) or Macintosh 3.5-inch diskette (plain text file [preferred], Word, or Write) or Sun-style QIC-24 cartridge tapes to:

> Hal Jespersen, Chair P1003.2
> POSIX Software Group
> 447 Lakeview Way
> Redwood City, CA 94062

Some degree of judgment is required in determining what actually changed in Draft 11.2. Use the diff marks as a guide, but they will frequently mark text that has no real normative changes. Please limit your objections to the actual changes: for example, if we change the `foo −x` option to `−y`, don't use that as an opportunity to object that we have no `−z` option. Your objection should only address why the `x` to `y` change is a problem. (We have been balloting for a long time now and it is time to tighten the consensus and finish this up.) If you find problems unrelated to changes, submit them as comments and they will be considered seriously in that category. Thanks for your cooperation on this.

Background on Balloting Procedures

The Balloting Group consists of over 160 technical experts who are members of the IEEE or the IEEE Computer Society; enrollment of individuals in this group has already been closed. There are also a few ''parties of interest'' who are not members of the IEEE or the Computer Society. Members of the Balloting Group are required to return ballots within the balloting period. Other individuals who may happen to read this draft are also encouraged to submit comments concerning this draft. The only real difference between members of the Balloting Group and other individuals submitting ballots is that *affirmative* ballots are only counted from Balloting Group members who are also IEEE or Computer Society members. (There are minimum requirements for the percentages of ballots returned and for affirmative ballots out of that group.) However, objections and nonbinding comments must be resolved if received from any individual, as follows:

(1) Some objections or comments will result in changes to the standard. This will occur either by the publication of a list of changes or by the republication of an entire draft. The objections/comments are reviewed by a team from the P1003.2 working group, consisting of the Chair, Vice Chair, the Chair of the TCOS Standards Subcommittee, and one or more Technical Reviewers. The Technical Reviewers each have subject matter expertise in a particular area and are responsible for objection resolution in one or more sections.

(2) Other objections/comments will not result in changes.

  (a) Some are misunderstandings or cover portions of the document (front matter, informative annexes, rationale, editorial matters, etc.) that are not subject to balloting.

  (b) Others are so vaguely worded that it is impossible to determine what changes would satisfy the objector. These are referred to as *Unresponsive*. (The Technical Reviewers will make a reasonable

effort to contact the objector to resolve this and get a newly worded objection.) Further examples of unresponsive submittals are those not marked as either *Objection* or *Comment* ; those that do not identify the portion of the document that is being objected to (each objec-
tion must be separately labeled); those that object to material in a 1
recirculation that has not changed and do not cite an unresolved 1
objection; those that do not provide specific or general guidance on what changes would be required to resolve the objection.

(c) Finally, others are valid technical points, but they would result in decreasing the consensus of the Balloting Group. (This judgment is made based on other ballots and on the experiences of the working group through almost five years of work and fifteen drafts preceding this one.) These are referred to as *Unresolved Objections* . Summaries of unresolved objections and their reasons for rejection are maintained throughout the balloting process, are circulated to members of the Balloting Group for their consideration, and are presented to the IEEE Standards Board when the final draft is
offered for approval. Unresolved objections are only circulated to 2
the balloting group when they are presented by members of the bal- 2
loting group or by parties of interest. Unsolicited correspondence 2
from outside these two groups may result in draft changes, but are 2
not recirculated to the balloting group members. 2

Please ensure that you correctly characterize your ballot by providing one of the following:

(1) Your IEEE member number

(2) Your IEEE Computer Society affiliate number

(3) If (1) or (2) don't apply, a statement that you are a ''Party of Interest''

Ballot Resolution

The general procedure for resolving ballots is:

(1) The balloting cuts off on 21 October 1991. This is a receipt date at the IEEE , not a postmark date. (Please do not telephone or FAX on 21 October 1991 and say that your specific comments will come later; late-arriving comments will not be considered as objections.) We will accept comments after that date, including direct e-mail to the working group officers or the Technical Reviewers, but they will be treated as comments only—not objections. And we don't guarantee a written response to these late submissions.

(2) The ballots are put online and distributed to the Technical Reviewers.

(3) If a ballot contains an objection, the balloter will be contacted individually by telephone, letter, or e-mail and the corrective action to be taken will be described (or negotiated). The personal contact will most likely not occur if the objection is very simple and obvious to fix or the balloter cannot be reached after a few reasonable attempts. Repeated failed

attempts to elicit a response from a balloter may result in an objection being considered unresponsive, based on the judgment of the working group chair. Once all objections in a ballot have been resolved, it becomes an affirmative ballot.

(4) If any objection cannot be resolved, the entire ballot remains negative.

(5) Once more than seventy-five percent of the ballots received (that had voted either affirmative or negative) have been turned affirmative, two lists are published to the entire balloting group: the detailed list of approved changes and the list of unresolved objections, along with our reasons for rejecting them. This is known as a *recirculation*. You have minimum of ten days (after an appropriate time to ensure the mail got through) to review these two lists and take one of the following actions:

  (a) Do nothing; your ballots will continue to be counted as we have classified them, based on items (3) and (4).

  (b) Explicitly change your negative ballot to affirmative by agreeing to remove all of your objections from the unresolved list.

  (c) Explicitly change your affirmative ballot to negative based on your disapproval of either of the two lists you reviewed. If an issue is not on one of the two lists, new objections about this are not allowed. Negative ballots that come in on recirculations cannot be cumulative. They shall repeat any objections that the balloter considers unresolved from the previous recirculation. Ballots that simply say ‘‘and all the unresolved objections from last time’’ will be declared unresponsive. Ballots that are silent will be presumed to fully replace the previous ballot, and all objections not mentioned on the most current ballot will be considered as successfully resolved.

(6) The list of changes will frequently be a new draft document with the changes integrated. This is not a requirement, however, and a small number of changes may prompt merely a change list approach to recirculation.

(7) A copy of all your objections and our resolutions will be mailed to you. You can receive the full package of all resolutions from all ballots by contacting the IEEE Standards Office (who will probably charge you for the copying involved). If you don't agree with one of our resolutions and haven't been contacted personally before you receive this list, please accept our apologies and submit a new ballot against the new draft during the recirculation period.

(8) If at the end of the recirculation period there remain greater than seventy-five percent affirmative ballots, and no new objections have been received, a new draft is prepared that incorporates all the changes. This draft and the unresolved objections list go to the IEEE Standards Board for approval. If the changes cause too many ballots to slip back into negative status, another resolution and recirculation cycle begins.

Balloting Guidelines

This section consists of guidelines on how to write and submit the most effective ballot possible. The activity of resolving balloting comments is difficult and time consuming. Poorly constructed comments can make that even worse.

We have found several things that can be done to a ballot that make our job more difficult than it needs to be, and likely will result in a less than optimal response to ballots that do not follow the form below. Thus it is to your advantage, as well as ours, for you to follow these recommendations and requirements.

If a ballot that significantly violates the guidelines described in this section comes to us, we will determine that the ballot is unresponsive, and simply ignore all the material in it.

Secondly, objections that don't contain a specification so that the correction to resolve the objection ‘‘can be readily determined’’ are also unresponsive and will be ignored.

(If we do recognize a ballot that is generally ‘‘unresponsive,’’ we will try to inform the balloter as soon as possible so he/she can correct it, but it is ultimately the balloter's responsibility to assure the ballot is responsive.)

Typesetting is not particularly useful to us. And please do not send handwritten ballots. Typewritten (or equivalent) is fine, and if some font information is lost it will be restored by the Technical Editor in any case. If you use `nroff`, you will include extraneous spacing and sometimes backspaces and overstrikes; if you really must use `nroff`, please turn off hyphenation and line adjusting:

```
.hy 0
.na
```

and run the output through `col -b` to remove all the overstrikes. (Also remember that backslashes and leading periods and apostrophes in your text will be treated impolitely by the `*roff` family). The ideal ballot is formatted as a ‘‘flat ASCII file,’’ without any attempt at reproducing the typography of the draft and without embedded control characters or overstrikes; it is then printed in Courier (or some other typewriter-like) font for paper-mailing to the IEEE Standards Office and simultaneously e-mailed to the working group Chair.

Don't quote others' ballots. Cite them if you want to refer to another's ballot. If more than one person wants to endorse the same ballot, send just the cover sheets and one copy of the comments and objections. [Note to Institutional Representatives of groups like X/Open, OSF, UI, etc.: this applies to you, too. Please don't duplicate objection text with your members.] Multiple identical copies are easy to deal with, but just increase the paper volume. Multiple almost-identical ballots are a disaster, because we can't tell if they are identical or not, and are likely to miss the subtle differences. Responses of the forms:

— ‘‘I agree with the item in <someone>'s ballot, but I'd like to see this done instead’’

— ‘‘I am familiar with the changes to `foo` in <someone>'s ballot and I would object if this change is [or is not] included’’ are very useful information to us. If we resolve the objection with the original balloter (the one whose ballot you are referencing), we will also consider yours to be closed, unless you specifically include some text in your objection indicating that should not be done.

Be very careful of ‘‘Oh, by the way, this applies <here> too’’ items, particularly if they are in different sections of the document that are likely to be seen by different reviewers. They are probably going to be missed! Note the problem in the appropriate section, and cite the detailed description if it’s too much trouble to copy it. The reviewers don’t have time to read the whole ballot, and only read the parts that appear to apply to them. Particularly where definitions are involved, even if the change really belongs in one section but the relevant content is in another, an extra cross-reference would be indicated. If you wish to endorse someone else’s ballot, either in whole or part, be specific about whether you will be automatically satisfied if they are satisfied. If you will not necessarily be satisfied if they are, your ballot could be deemed unresponsive because it does not give achievable conditions under which your ballot could be converted to affirmative. You then must give the conditions under which you would be satisfied as well. If you would be satisfied in some areas and not in others, it is best to specifically point to each specific objection in the ballot you point to, giving the conditions for each.

Please consider this a new ballot that should stand on its own. Please do not make backward references to your ballots for previous drafts—include all the text you want considered here, because the Technical Reviewer may not have your old ballot. And, the old section and line numbers won’t match up anyway. If one of your objections was not accepted exactly as you wanted, it will not be useful to send in the exact text you sent before; read the nearby Rationale section and come up with a more compelling (or clearly-stated) justification for the change.

Please be very wary about global statements, such as ‘‘all of the arithmetic functions need to be defined more clearly.’’ Unless you are prepared to cite specific instances of where you want changes made, with reasonably precise replacement language, your ballot will be considered unresponsive.

Ballot Form

The following form is recommended. We would greatly appreciate it if you sent the ballot in electronic form in addition to the required paper copy. Our policy is to handle all ballots online, so if you don’t send it to us that way, we have to type it in manually. For the last POSIX.2 ballot, only one or two balloters could not accommodate us on this and thus we had very little typing to do. See the first page of this Annex for the addresses and media. As you’ll see from the following, formatting a ballot that’s sent to us online is much simpler than a paper-only ballot.

The ballot should be page-numbered, and contain the name, e-mail address, and phone number(s) of the objector(s). (If you send us only a paper copy, make sure this information appears on every page; electronic ballots just need it once, in the beginning.) The lines before the first dashed line are a page header, and should only appear once on each page. Please leave adequate (at least one inch) margins on both sides. Each objection/comment/editorial comment should be sequentially numbered, not in individual ranges [i.e., not Objection #1, Comment #1]

Since we deal with the ballots online, there is no longer any requirement to put only one objection or section per page.

Don't format the ballot as a letter or document with its *own* section numbers. These are simply confusing. As shown below, it is best if you cause each objection and comment to have a sequential number that we can refer to amongst ourselves and to you over the phone. Number sequentially from 1 and count objections, comments, and editorial comments the same; don't number each in its own range. If you don't do this, we'll number them ourselves, but you won't know what numbers we're using.

Please precede each objection/comment with a little code line (if you don't, we'll have to do it ourselves):

@ *<section>*.*<clause>* *<code>* *<seqno>*

where:

@ — At-sign in column 1 (which means no @'sin any other column 1's).

<section> — The major section (chapter or annex) number or letter in column 3. Use zero for Global or for something, like the frontmatter, that has no section or annex number.

<clause> — The clause number (second-level header). Please do not go deeper than these two levels. In the text of your objection or comment, go as deep as you can in describing the location, but this code line uses two levels only.

<code> — One of the following lowercase letters, preceded and followed by spaces:

- o Objection.
- c Comment or Editorial Comment.

<seqno> — A sequence number, counting all objections and comments in a single range.

Objection:

```
Balloter Name              (202)555-1212       page  x of nn.
E-Mail Address             FAX:   Fax Number
Balloter2 Name                    (303)555-1213
E-Mail Address2            FAX:   Fax Number2
-----------------------------------------------------------------
@ x.y o seq#
<Seq#> Sect x.y OBJECTION. page xxx, line zzz:
```

`Problem:`

A clear statement of the problem that is observed, sufficient for others to understand the nature of the problem. Note that you should identify problems by section, page, and line numbers. This may seem redundant, but if you transpose a digit pair, we may get totally lost without a cross-check like this. Use the line number where the problem starts, not just where the section itself starts; we sometimes attempt to sort objections by line numbers to make editing more accurate. If you are referring to a range of lines, please don't say ''lines 1000ff;'' use a
real range so we can tell where to stop looking. If you have access to the online 2
versions of a balloting draft, please do not send in a ballot that refers to the page 2
numbers in the `nroff` output version; use only the line and page numbers found 2
in the printed draft or the online PostScript draft. We will really love you if you 2
can manage to include enough context information in the problem statement (such as the name of the utility) so we can understand it without having the draft in our laps at the time. (It also helps you when we e-mail it back to you.) If you are objecting to an action in the Unresolved Objections List, use the section/page/line number reference for the appropriate place in the standard;
don't refer to the UOL except to cite its number and for clarification of your points. 1

`Action:`

A precise statement of the actions to be taken on the document to resolve the objection above, which if taken verbatim will completely remove the objection.

If there is an acceptable range of actions, any of which will resolve the problem for you if taken exactly, please indicate all of them. If we accept any of these, your objection will be considered as resolved.

If the Action section is omitted or is vague in its solution, the objection will be reclassified as a nonbinding comment. The Technical Reviewers, being human, will give more attention to Actions that are well-described than ones that are vague or imprecise. The best ballots of all have very explicit directions to substitute, delete, or add text in a style consistent with the rest of the document, such as:

```
Delete the sentence on lines 101-102:
      "The implementation shall not ... or standard error."
On line 245, change "shall not" to "should not".
After line 103, add:
       -r    Reverse the order of bytes read from the file.
```

Some examples of poorly-constructed actions:

```
Remove all features of this command that are not supported by BSD.
Add -i.
Make this command more efficient and reliable.
Use some other flag that isn't so confusing.
I don't understand this section.
Specify a value--I don't care what.
```

Objection Example:

```
Hal Jespersen             (415) 364-3410            page  3 of 17.
UUCP: hlj@Posix.COM  FAX:   (415) 364-4498
------------------------------------------------------------------
@ 2.6 o 23
23. Sect 2.6 OBJECTION. page 77, line 1217:

Problem:

The EDITOR environment variable is not used as stated
in my company.  This description would cause hundreds
of my shell scripts to break.

Action:

Change the first sentence on line 1217 to:

      The e-mail address of the editor of the user's
      favorite POSIX standard.

-----------------------
@ 3.1 o 24
24. Sect 3.1.6 OBJECTION. page 123, line 17:

Problem:

I support UO 3.01-999-6 concerning the objection to the               1
definition of "operator".                                             1
This definition would cause great hardship to the users               1
of the systems I develop.                                             1
I feel your rationale for rejection was inappropriate                 1
because you overlooked the following technical points [etc.]...       1

Action:

Change the term "operator" to "operation-symbol" in this
definition and globally throughout Section 3.
```

Comment:

```
-------------------------------------------------------------------
@ x.z c seq#
<Seq#> Sect x.z COMMENT. page xxx, line zzz:
```

A statement of a problem that you might want to be resolved by the reviewer, but which does not in any way affect whether your ballot is negative or positive. The form for objections is not required, but it increases the probability that your comment will have an effect on the final document.

Although there may be questions to you or responses on the topic, no changes in the drafts are required by a comment, although it will be looked at to determine whether the concern should be addressed. It is possible to abuse this rule and label all of your comments as objections, but it is a significant disservice to the individuals who are volunteering their time to address your concerns.

Remember that any issue concerning the pages preceding page 1 (the Frontmatter), Rationale text with shaded margins, Annexes, NOTES in the text, footnotes, or examples will be treated as a nonbinding comment whether you label it that way or not, but it would help us if you'd label it correctly.

Editorial Comment:

```
-------------------------------------------------------------------
@ x.z c seq#
<Seq#> Sect x.z EDITORIAL COMMENT. page xxx, line zzz:
```

These are for strictly editorial issues, where the technical meaning of the document is not changed. Examples are: typos; misspellings; English syntax or usage errors; appearances of lists or tables; arrangement of sections, clauses, and subclauses (except where the location of information changes the optionality of a feature). Marking these as comments but indicating that they are editorial speeds the process.

Please be aware that after balloting concludes the document will be subjected to more sets of editors at the IEEE and ISO who are empowered to make broad editorial changes and rewording (for example, to get the text ready for translation into French.)

Thank you for your cooperation in this important balloting process.

Hal Jespersen

Identifier Index

Alphabetic Topical Index

A

B

C

D

E

F

G

H

J

K

L

M

O

P

Q

T

V

W

Acknowledgm ents

We wish to thank the following organizations for donating significant computer, printing, and editing resources to the production of this standard: Amdahl Corporation, AT&T, Concurrent Computer Corporation, the POSIX Software Group, UniSoft Corporation, and the X/Open Group.

This document was also approved by ISO/IEC JTC 1/SC22/WG15 as ISO/IEC 9945-2:199x. The IEEE wishes to thank the advisory groups of the National Bodies participating in WG15 for their contributions: Austria, Belgium, Canada, Denmark, France, Germany, Japan, Netherlands, United Kingdom, USA, and USSR.

The IEEE also wishes to thank the delegates to WG15 for their contributions:

AUSTRIA
Gerhard Schmitt
Wolfgang Schwabl

CANADA
Joe Cote
Patrick Dempster
George Kriger
Bernard Martineau
Major Douglas J. Moore
Arnie Powell
Paul Renaud
Richard Sniderman

CEC
Phil Bertrand
Manuel Carbajo
Michel Colin

DENMARK
Peter E. Cordsen
Isak Korn
Keld Simonsen
Claus Tondering

FINLAND
Jikka Haikala

FRANCE
Pascal Beyles
Christophe Binot
Claude Bourstin
Jean-Michel Cornu
Yves Delarue
Eric Dumas
Maurice Fathi
Gerald Krummeck
Herve Schauer
Hubert Zimmerman

GERMANY
Ron Elliot
Helmut Stiegler
Claus Unger
Rainer Zimmer

IRELAND
Hans-Jurgen Kugler

JAPAN
Hiromichi Kogure
Shigekatsu Nakao
Yasushi Nakahara
Nobuo Saito

NETHERLANDS
J. Van Katwijk
Willem Wakker
H.J. Weegenaar

SWEDEN
Mat Linder

UK
Nigel Bevan
Cornelia Boldyreff
Dave Cannon
Don Chacon
Dominic Dunlop
David Flint
Don E. Folland
Martin Kirk
Neil Martin
Brian Meek
Kevin Murphy
Ian Newman
Philip Rushton

USA
Robert Bismuth
Steven L. Carter
Terence S. Dowling
Ron Elliott
Dale Harris
John Hill
James D. Isaak
Hal Jespersen
Roger J. Martin
Shane McCarron
Barry Needham
Donn S. Terry
Alan Weaver

USSR
V. Koukhar
Ostapenko Georgy Pavlovich

Also we wish to thank the organizations employing the members of the Working Group and the Balloting Group for both covering the expenses related to attending and participating in meetings, and donating the time required both in and out of meetings for this effort.

3M Company
ACE Associated Computer Experts b.v.
Aeon Technologies, Inc.
Mallinckrodt Institute
Martin Marietta Data Systems
Masscomp

Alis Development
Amdahl Corporation *
Apollo Computer Inc.
Apple Computer Inc.
Ardent Computer
AT&T *
AT&T Bell Laboratories
AT&T UNIX Pacific Co., Ltd.
Axon Data Information Systems
BBN Communications Corporation
Bell Communications Research
BNR, Inc.
Bolt Beranek & Newman
BP Research International
British Telecom
British Telecom Research Labs
Charles River Data Systems *
Chemical Abstracts Service
Chorus Systemes
Commission of the European Communities
Computer X, Inc.
Concurrent Computer Corporation *
Control Data Corporation *
Convergent Technologies
Convex Computer Corporation
Cray Research, Inc. *
Cyber-Dyne, Inc.
Datapoint Corporation
Data General Corporation
Digital Equipment Corporation *
Digital ETU
Douglas Aircraft Company
Electrospace Systems, Inc.
Emerging Technology Group Inc.
Emory University Computing Center
Encore Computer
ETA Systems, Inc.
Federal Judicial Center
Ford Motor Company
Free Software Foundation
General Electric Corporation
Georgia Institute of Technology
Gilbert International Inc.
Gould CSD
Harris Corporation
HCR Corporation
Hewlett-Packard Company *
Honeywell Bull, Inc.
HQ USAISC
Hughes Aircraft Co.
IBM Corporation *
IBM Japan
IBM Systems Integration Division
Icon International, Inc.
Intel
Interactive Systems Corporation
Ironwood Software
KAIST
Mercury Computer Systems
Microsoft Corporation
Mindcraft, Inc.
Mitsubishi Electric Corporation
Mortice Kern Systems Inc.
Motorola Inc.
Myrias Research Corporation
NAPS International
NASA-KSC
National Institute of Standards and Technology
Naval Postgraduate School
NCR Corporation *
Northern Telecom, Inc.
Novell, Inc.
Ohio State University
Pacific Marine Tech
POSIX Software Group
PRC
Prime Computer, Inc.
R&D Associates
Rabbit Software Corporation
ROLM Mil Spec Computers
Sandia National Laboratory *
Santa Cruz Operation Inc.
Saudia National Labs
Sequent
Shia Systems Inc.
Simpact Associates, Inc.
SoHar
Sphinx Ltd.
St. Lawrence College
Stellar Computer Inc.
Sun Microsystems, Inc. *
Syntactics
Tandem Computers Inc.
Technical University of Vienna
Tektronix, Inc. *
Texas Instruments
The Instruction Set Ltd.
The MITRE Corp
Toshiba Corporation
UFP b-GRC
UniForum *
UniSoft Corporation
UniSoft Ltd.
Unisys Corporation *
University Of California, Berkeley
University of Hong Kong
University of Indonesia
University of Maryland
University of Texas at Arlington
University of Utah
University of Victoria
University of Vienna
USAF
USAISEC
USENIX Association*
US Army

Lachman Associates, Inc.
Lawrence Livermore National Lab
Loral Rolm MilSpec Computers
Lotus Development Corporation
Mahavishi International University
Whitesmiths, Ltd.
Woods Hole Oceanographic Inst.
US Army Ballistic Research Lab.
US Army Computer Engineering Center
US West Advanced Technologies
Videoton
Wang Laboratories
X/Open Company Ltd.
XIOS Systems Corporation

In the preceding list, the organizations marked with an asterisk (*) have hosted 1003 Working Group meetings since the group's inception in 1985, providing useful logistical support for the ongoing work of the committees.