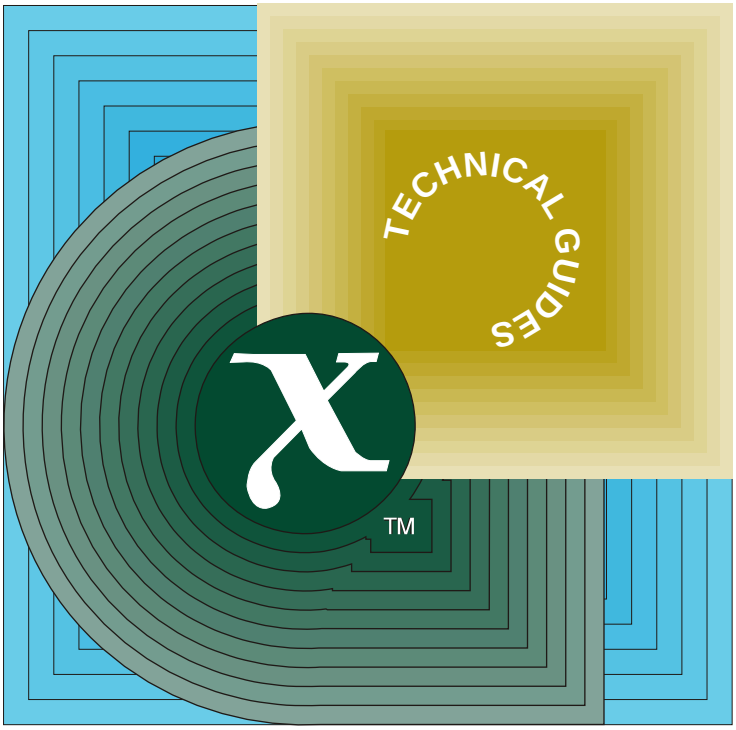
Guide

XPG3-XPG4 Base Migration Guide
Version 2



THE Open GROUP

[This page intentionally left blank]

X/Open Guide

XPG3-XPG4 Base Migration Guide, Version 2

X/Open Company Ltd.

X/Open Guide

XPG3-XPG4 Base Migration Guide, Version 2

ISBN: 1-85912-156-X

X/Open Document Number: G501

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

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

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

or by Electronic Mail to:

XoSpecs@xopen.org

Contents

List of Examples

List of Tables

Preface

X/Open

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

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

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

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

X/Open Technical Publications

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

There are two types of X/Open specification:

- *CAE Specifications*

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

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

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

- *Preliminary Specifications*

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

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

In addition, X/Open publishes:

- *Guides*

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

- *Technical Studies*

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

- *Snapshots*

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

Versions and Issues of Specifications

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

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

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

Corrigenda

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

The reader of this document is advised to check periodically if any Corrigenda apply to this publication. This may be done in any one of the following ways:

- anonymous ftp to ftp.xopen.org
- ftpmail (see below)
- reference to the Corrigenda list in the latest X/Open Publications Price List.

To request Corrigenda information using ftpmail, send a message to ftpmail@xopen.org with the following four lines in the body of the message:

```
open
cd pub/Corrigenda
get index
quit
```

This will return the index of publications for which Corrigenda exist. Use the same email address to request a copy of the full corrigendum information following the email instructions.

This Document

This document is a Guide (see above). It is a companion to the set of CAE specifications that define the X/Open System Interface (XSI) Operating System requirements:

- System Interface Definitions, Issue 4, Version 2 (XBD)
- Commands and Utilities, Issue 4, Version 2 (XCU)
- System Interfaces and Headers, Issue 4, Version 2 (XSH).

This document provides detailed guidance on applications migration from a number of perspectives:

- from Issue 3 conforming systems to Issue 4 of the XSI
- from traditional UNIX operating systems environments to systems conforming to Issue 4, Version 2 of the XSI
- from traditional UNIX operating systems environments to systems conforming to Issue 4 of the XSI.

It is intended for application developers, who are expected to be experienced C-language programmers or familiar with the shell command language and utilities. It is also intended for interactive users of the shell interpreter. It provides useful information for implementors.

This guide is structured as follows:

- General Information
  - — Chapter 1 is an introduction.
  - — Chapter 2 discusses global changes affecting many of the functions, macros, external variables and utilities; much of this material relates to the XBD specification.
- Commands and Utilities Migration
  - — Chapter 3 discusses the changes in the shell command language.
  - — Chapter 4 details specific changes for individual utilities; unlike the XCU specification, however, the utilities that are withdrawn in Issue 4 are included in this chapter, sorted alphabetically.
- System Interfaces and Headers Migration
  - — Chapter 5 describes the compilation environment, error numbers and interprocess communication.
  - — Chapter 6 is a quick reference guide to the functions, macros and external variables defined in Issue 4, Version 2.
  - — Chapter 7 details specific changes for individual interfaces.
  - — Chapter 8 details specific changes for individual headers.
- C-language Migration
  - — Chapter 9 is a brief introduction to C-language migration.
  - — Chapter 10 describes function prototypes and how to use them.
  - — Chapter 11 discusses promotions.
  - — Chapter 12 describes tokenisation and preprocessing.
  - — Chapter 13 describes the use of type qualifiers, incomplete types, compatible types and composite types.
  - — Chapter 14 describes expressions.
  - — Chapter 15 describes multi-byte characters and wide-character codes.
  - — Chapter 16 describes standard headers and reserved names.

Comprehensive references are available in the index.

Typographical Conventions

The following typographical conventions are used throughout this document:

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

  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. Brackets shown in this font, `[ ]`, are part of the syntax and do *not* indicate optional items. In syntax the `|` symbol is used to separate alternatives, and ellipses (`. . .`) are used to show that additional arguments are optional.
- **`Bold fixed width`** font is used to identify brackets that surround optional items in syntax, **`[ ]`**, and to identify system output in interactive examples.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- Ranges of values are indicated with parentheses or brackets as follows:
  - — (a,b) means the range of all values from a to b, including neither a nor b
  - — [a,b] means the range of all values from a to b, including a and b
  - — [a,b) means the range of all values from a to b, including a, but not b
  - — (a,b] means the range of all values from a to b, including b, but not a
- Shading is used to identify extensions or warnings as detailed in Section 1.5 on page 9.

Acknowledgements

X/Open gratefully acknowledges:

- AT&T for permission to reproduce portions of its copyrighted System V Interface Definition (SVID) and material from the UNIX System V Release 2.0 documentation.
- The Institution of Electrical and Electronics Engineers, Inc. for permission to reproduce portions of its copyrighted IEEE Std 1003.2/D12.
- The IEEE Computer Society's Technical Committee on Operating Systems and Application Environments (TCOS), whose standards contributed to our work.
- The ANSI X3J11 Committees.
- Hewlett-Packard Company, International Business Machines Corporation, Novell Inc., The Open Software Foundation, and Sun Microsystems, Inc., for their work in developing the X/Open UNIX Extension and sponsoring it through the X/Open Direct Review (Fast-track) process.
- The /usr/group Standards Committee, whose standard contributed to this work.
- The UniForum (formerly /usr/group) Technical Committee's Internationalization Subcommittee for work on internationalised regular expressions.

Trade Marks

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

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

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

X/Open® is a registered trade mark, and the ‘‘X’’ device is a trade mark, of X/Open Company Limited.

Referenced Documents

The following X/Open documents are referenced in this guide:

How to Brand — What to Buy
: The X/Open Branding Programme, How to Brand — What to Buy, February 1995 (ISBN: 1-85912-084-9, X951).

Internationalisation Guide
: X/Open Guide, July 1993, Internationalisation Guide, Version 2 (ISBN: 1-872630-02-4, G304).

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

Issue 2
: Consists of:

  - X/Open Portability Guide, Volume 1, January 1987, XVS Commands and Utilities (ISBN: 0-444-70174-5).
  - X/Open Portability Guide, Volume 2, January 1987, XVS System Calls and Libraries (ISBN: 0-444-70175-3).

Issue 3
: Consists of:

  - Commands and Utilities, Issue 3
  - System Interfaces and Headers, Issue 3.

Issue 4
: Consists of:

  - System Interface Definitions, Issue 4
  - Commands and Utilities, Issue 4
  - System Interfaces and Headers, Issue 4.

Issue 4, Version 2
: Consists of:

  - System Interface Definitions, Issue 4, Version 2
  - Commands and Utilities, Issue 4, Version 2
  - System Interfaces and Headers, Issue 4, Version 2.

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

Networking Services, Issue 4
: X/Open CAE Specification, August 1994, Networking Services, Issue 4 (ISBN: 1-85912-049-0, C438).

X/Open C
: Chapters 1 to 4 of X/Open Specification, 1988, 1989, February 1992, Programming Languages, Issue 3 (ISBN: 1-872630-39-1, C214); this specification was formerly X/Open Portability Guide, Volume 4, August 1988 (ISBN: 0-13-685868-6, XO/XPG/89/005).

X/Open Curses, Issue 4
: X/Open CAE Specification, December 1994, X/Open Curses, Issue 4 (ISBN: 1-85912-077-6, C437).

XBD, Issue 3
: X/Open Specification, 1988, 1989, February 1992, Supplementary Definitions, Issue 3 (ISBN: 1-872630-38-3, C213); this specification was formerly X/Open Portability Guide, December 1988, Volume 3, (ISBN: 0-13-685850-3, XO/XPG/89/004).

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

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

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

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

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

XSH, Issue 3
: X/Open Specification, February 1992, System Interfaces and Headers, Issue 3 (ISBN: 1-872630-37-5, C212); this specification was formerly X/Open Portability Guide, Issue 3, Volume 2, January 1989, XSI System Interface and Headers (ISBN: 0-13-685843-0, XO/XPG/89/003).

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

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

The following non-X/Open documents are referenced in this guide:

ANSI C
: American National Standard for Information Systems: Standard X3.159-1989, Programming Language C.

AWK Programming Language
: The AWK Programming Language by Aho, Kernighan and Weinberger.

C Programming Language
: The C Programming Language (First Edition), by Kernighan and Ritchie.

FIPS 151-2
: Proposed Federal Information Procurement Standards (FIPS) 151-2.

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

ISO C
: ISO/IEC 9899:1990, Programming Languages — C (technically identical to ANSI standard X3.159-1989).

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

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

MSE working draft
: Working draft of ISO/IEC 9899:1990/Add3:draft, Addendum 3 — Multibyte Support Extensions (MSE) as documented in the ISO Working Paper SC22/WG14/N205 dated 31 March 1992.

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

POSIX.1b
: IEEE Std 1003.1b-1993, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] — Amendment 1: Realtime Extension.

POSIX.2
: IEEE Std 1003.2-1992, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities.

POSIX.2a
: IEEE Std 1003.2a-1992, Standard for Information Technology — Portable Operating System Interface (POSIX) — Part 2: Shell and Utilities — Amendment 1: User Portability Extension.

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

XPG3-XPG4 Base Migration Guide, Version 2

Part 1:

General Information

X/Open Company Ltd.

Chapter 1

Introduction

This document has been created to aid developers migrating applications source code from XPG3 Base branded systems and traditional UNIX systems to XPG4 Base and X/Open UNIX branded systems. It updates the original XPG3-XPG4 Base Migration Guide.

The X/Open Portability Guide (XPG) evolved into the X/Open Common Applications Environment (CAE) between Issue 3 and Issue 4. Issue 3 of the Portability Guide (XPG3) aligned with the emerging ISO POSIX standards, and was published as Issue 4 in July 1992. While the overall specifications structure had become the X/Open CAE, XPG4 Base was used as an easy way to refer to the three volumes that formed the core programming portability specifications. The XPG3-XPG4 Base Migration Guide was published to help programmers migrate applications developed on XPG3 Base branded systems, to XPG4 Base branded systems.

These three core specifications were re-published in September 1994 as Issue 4, Version 2, as part of the Single UNIX Specification, and contain a considerable number of new interfaces and new functionality. They are published as a new *version* rather than as a new *issue*, as they are a proper superset of Issue 4, adding new interfaces and functionality, but not modifying or removing any Issue 4 material. (Networking Services, Issue 4 and X/Open Curses, Issue 4 form the rest of the Single UNIX Specification. These two documents are not discussed in the context of this guide.) The Single UNIX Specification was developed to simplify the porting of applications developed to run on traditional UNIX systems.

Systems and components can still be branded to XPG3 Base, but the XPG4 Base brand offers the customer and user significant additional capability. The XPG4 UNIX brand, as described by the Single UNIX Specification, offers applications developers additional advantages when porting existing applications developed on a traditional UNIX implementation.

This chapter explains the objective of this guide, identifies the documentation relevant to the X/Open Operating System Interface (XSI), and explains its relationship to formal and emerging standards. This chapter also explains how it affects application developers and interactive users. A brief discussion of XPG4 profiles closes the chapter.

Throughout this guide, the term *system interfaces* covers the functions, macros and external variables specified in the XSH specification. The term *interface* is used generically to refer to one of the system interfaces or a utility defined in the XCU specification.

1.1 Objectives

The objective of this guide is to provide information for application developers so that:

- New applications can be developed for maximum portability between XSI-conformant systems.
- Existing applications that conform to Issue 3 can be updated to conform with the facilities and interfaces defined in the XSI specification set.
- Existing applications that were developed on a traditional UNIX system can be ported with a minimum of work.
- Existing applications that were developed on a traditional UNIX system can be updated to be more portable to systems that conform to the previous version of the XSI specification set.

Note: Issue 4 is designed such that strictly complying Issue 3 programs should continue to work without modification, except as noted in Section 1.5 on page 9. However, a number of interfaces have been added in areas where hitherto there were no standards, and individual interfaces may have been extended to formalise common-use capability. This is especially true for the X/Open UNIX Extension. Also, a number of interfaces that are defined as optional in Issue 3 are mandatory in Issue 4. Thus, their availability is ensured on all implementations that conform to Issue 4. XPG4 UNIX brand-conformant systems require all feature groups considered optional for XPG4 Base 95 conformance.

1.2 XSI Documentation — XPG3, XPG4 and the Single UNIX Specification

The XSI specification for XPG3 Base is contained in the following CAE specifications:

- Supplementary Definitions, Issue 3 (formerly entitled XSI Supplementary Definitions, Volume 3 of the X/Open Portability Guide, Issue 3)
- Commands and Utilities, Issue 3 (formerly entitled XSI Commands and Utilities, Volume 1 of the X/Open Portability Guide, Issue 3)
- System Interfaces and Headers, Issue 3 (formerly entitled XSI System Interfaces and Headers, Volume 2 of the X/Open Portability Guide, Issue 3)
- Chapters 1 to 4 in Programming Languages, Issue 3 (formerly entitled Programming Languages, Volume 4 of the X/Open Portability Guide, Issue 3)

The XSI specification for XPG4 Base and XPG4 UNIX comprises:

- System Interface Definitions, Issue 4, Version 2 (XBD)
- Commands and Utilities, Issue 4, Version 2 (XCU)
- System Interfaces and Headers, Issue 4, Version 2 (XSH).

These documents cover the XSI portion of the Single UNIX Specification.

The X/Open C Language remains documented in Programming Languages, Issue 3. XPG4 Base and XPG4 UNIX require C-language support according to the ISO C standard and, to enable migration of applications developed on XPG3 Base branded systems, requires support for the X/Open C Language as defined for XPG3 Base.

This guide is a companion to the Issue 4, Version 2 XSI specification set.

1.3 Conformance to Standards

XSI Issue 3 is fully compliant with IEEE Std 1003.1-1988 (POSIX.1).

XSI Issue 4 is fully compliant with:

- ISO / IEC 9945-1: 1990 (POSIX-1)
- ISO / IEC 9945-2: 1993 (POSIX-2)
- ISO / IEC 9899: 1990, Programming Languages — C (ISO C)
- FIPS 151-2.

The POSIX-1 standard is an updated version of the IEEE Std 1003.1-1988 (POSIX.1), which defines the system application programming interface (API) of the Portable Operating System Interface (POSIX) developed by IEEE working group P1003.1. The interfaces are defined in terms of C-language bindings.

The XSH specification also contains parts of the POSIX-2 standard Shell and Utilities, which are deemed by X / Open to be more appropriate to the XSH specification than to the XCU specification. The POSIX-2 standard is a full-use standard, and is equivalent to IEEE Std 1003.2-1992 (POSIX.2) and IEEE Std 1003.2a-1992 (POSIX.2a).

The Federal Information Processing Standards (FIPS) are a series of U.S. government procurement standards managed and maintained on behalf of the U.S. Department of Commerce by the National Institute of Standards and Technology (NIST). FIPS 151-2 is a profile of the POSIX-1 standard, incorporating its text by reference, and mandating selected optional behaviours and setting certain limits. The main impact of alignment with FIPS 151-2 is that various optional elements of XSI Issue 3 are mandatory in XSI Issue 4, for example:

- job control
- supplementary process groups
- saved set-user-IDs and saved set-group-IDs
- {_POSIX_CHOWN_RESTRICTED}
- {_POSIX_VDISABLE}
- {_POSIX_NO_TRUNC}.

Differences that this alignment causes in individual interface definitions are highlighted in the XSH specification and the XCU specification, and marked with a ‘‘FIPS’’ portability code in the margin.

The ISO C standard is equivalent to the ANSI C standard, which defines a standard for the C programming language. This standard was originally developed by the ANSI X3J11 working group. One of the major effects of the ISO C standard on XSI Issue 4 is that all interfaces are now defined in terms of function prototypes.

The XSH specification is also aligned with:

- ISO / IEC 9899: 1990 / Add3: draft (Multibyte Support Extensions).

Issues of timing caused the XSH specification to be aligned with a draft of this standard, which had not been ratified at the time. There are small areas where the XSH specification does not align with the now ratified addendum, because of this timing. These areas will be corrected in a future version or issue of the XSH specification. These differences are pointed out in their respective descriptions.

1.3.1 C Language in an Issue 4 Environment

ISO C is the language specified in the ISO C standard. Common Usage C refers to the C Language before standardisation. Differences exist between ISO C and Common Usage C. XPG1 to XPG3 define Common Usage C as the XSI C programming language, whereas XPG4 requires support for both ISO C and X/Open C, which is a Common Usage C.

Differences also exist between the C library definitions published in successive issues of the Portability Guide. Issue 1 and Issue 2 derived their C library definitions from SVID Issue 1; Issue 3 retains the same function syntax as Issue 2 but header usage is aligned with the library definition published in ANSI standard X3.159-1989, Programming Language C (ANSI C). Issue 4 is aligned with the library definition published in the ISO C standard.

Information about the portability of applications between Issue 3 and Issue 4 is contained in this guide, in terms of language and C library usage. In particular, Part 3 identifies where the types of arguments and return values of individual functions are changed to align with ISO C.

Application programmers are warned that the use of earlier definitions of C library functions need not be fully supported in Issue 4 environments. Information about migration to the use of ANSI C headers can be found in the **APPLICATION USAGE** and **CHANGE HISTORY** sections of System Interfaces and Headers, Issue 3.

1.4 Portability

An important aspect of this document is to describe changes required to applications that conform to Issue 3 to allow varying degrees of portability:

- to all POSIX (POSIX-2 standard and POSIX-1 standard) systems in the future
- to all POSIX systems, but using obsolescent features that may be withdrawn in the future
- to all X/Open branded systems
- to all systems of a particular vendor.

This document does not deal with these classes of portability as separate issues. The documents in the XSI specification set contain warnings highlighting non-portable features. These take the form of markings in the margins, coupled with text shading. If the entire **SYNOPSIS** section for the interface or utility is shaded, then the interface or utility in its entirety is an extension beyond the base formal standards. Some interfaces are collected together into feature groups, that need not be implemented on a branded system, depending upon the particular XPG4 profile to which the system is branded. See Section 1.7 on page 11.

A maximally portable application (one portable to all POSIX-based systems for the foreseeable future) cannot use any of the features with marginal notations and background shading in the XSI specification set. Judicious use can be made of features marked OB (obsolescent), EX (an X/Open extension to POSIX), FIPS (an extension to conform to the FIPS), or JC (an extension involving the presence of the job control feature on all X/Open systems), but OB may require changes in a few years and EX, FIPS and JC might not be portable to systems that are not conformant to an XPG4 profile. Using features marked UX (an X/Open UNIX Extension) may limit portability to systems that are X/Open Single UNIX Specification-conformant. Using any of the other marked features limits portability to a subset of XPG4 Base-conformant systems. These specifications may be supported differently by systems branding to different XPG4 profiles. For example, an XPG4 Base 95 branded system may or may not support interfaces that comprise certain feature groups (for example, SHARED MEMORY), while XPG4 UNIX systems support all of the interfaces.

1.5 Existing Shell Applications

There are a few shell applications (or C applications that call the shell) that will cease to operate when a system is converted to Issue 4 conformance. This is because previously valid input to the shell command interpreter or to a utility has changed in meaning to correct a problem in the historical design, to close a security loophole, to extend the system for internationalisation, or to converge the differing syntaxes from systems such as System V, BSD, and so on. These applications must be modified before a conversion to Issue 4 compliance is considered complete.

CW When changes in Issue 4 commands and utilities cannot be made in a forward-compatible way from Issue 3, they are identified in this document with background shading and a marginal notation of CW (Compatibility Warning), as in this example.

The CW notation is not used to protect applications that rely on features or behaviour not described in Issue 3 or on those marked with portability warnings such as MV (marginal value), OF (output format incompletely specified), PI (portability inconsistent) or UN (possibly unsupportable feature).

1.6 Guidelines for Interactive Users

A user who follows the guidelines given for shell applications should be able to change from one system to another without learning new conventions and rules. Such a user should observe the same marginal warning markings that application writers do.

1.7 XPG4 Profiles

This guide describes the differences between issues of the XSI specification set as it has evolved. These specifications form the basis for *component definitions*, which in turn are put together to form *profile definitions*. Systems that are branded, are branded against these profile definitions. While a developer uses the specifications when developing and porting applications programs, the systems on which they work are branded to X/Open profiles. Branded products all have *conformance statements* that describe such items as which feature groups are implemented on a system. There are several key XPG4 profiles, all of which are completely defined in Part 3, Profile Definitions of How to Brand — What to Buy. All components used in profile definitions are defined in Part 2, Component Definitions of How to Brand — What to Buy.

1.7.1 XPG4 Base Profile

The original XPG4 Base profile, published in 1992, describes a fully functional environment for portable applications development. It requires full conformance to the XPG4 Internationalised System Calls and Libraries component, as described by System Interfaces and Headers, Issue 4, and a soft brand for the XPG4 Commands and Utilities component, as described by Commands and Utilities, Issue 4. This component can be anywhere along the migration path from XPG3 Commands and Utilities to a fully conforming XPG4 Commands and Utilities. Feature groups identified in System Interfaces and Headers, Issue 4 are optionally implemented, with the exception of the POSIX.2 C-language Binding feature group.

Required components:

- XPG4 Internationalised System Calls and Libraries
- XPG4 Commands and Utilities
- XPG4 C Language.

1.7.2 XPG4 Base 95 Profile

This is the current enhanced XPG4 Base profile requiring full conformance to the XPG4 Commands and Utilities V2 component definition. Feature groups identified in System Interfaces and Headers, Issue 4 are optionally implemented, with the exception of the POSIX.2 C-language Binding feature group.

Required components:

- XPG4 Internationalised System Calls and Libraries
- XPG4 Commands and Utilities V2
- XPG4 C Language.

1.7.3 XPG4 UNIX Profile

This profile is a superset of XPG4 Base 95, and describes a platform supporting the additional functions for applications portability for programs originally developed on traditional UNIX systems. All feature groups identified in System Interfaces and Headers, Issue 4 are required to be implemented. (This requirement does not apply to interfaces in the Encryption feature group, as U.S. Federal government export restrictions may prevent a vendor from shipping this functionality.)

Required components:

- XPG4 Internationalised System Calls and Libraries Extended

- XPG4 Commands and Utilities V2
- XPG4 C Language
- XPG4 Transport Service (XTI)
- XPG4 Sockets
- XPG4 Internationalised Terminal Interfaces.

A brand is further qualified by an attribution. In order to clearly identify to purchasers the exact specification to which a branded product conforms, an additional attribution is required when reference is made to the X/Open brand and branded products, or to the trade mark and the product registration.

XPG4 profile-related attributions are:

Profile Name	Attribution
XPG4 Base	Base
XPG4 Base 95	Base 95
XPG4 UNIX	UNIX 95
UNIX 93†	UNIX 93

Table 1-1 X/Open XPG4 Profiles and Brand Attributions

So, depending upon the profile definition to which a system was branding, they would claim: ''ProductName Version X.X is an X/Open <Attribution> branded product''. If the system was branded to the complete X/Open Single UNIX Specification definition, described by the XPG4 UNIX Profile, they would claim: ''ProductName Version X.X is an X/Open UNIX 95 branded product''.

---

† UNIX 93 is an interim brandable profile. To brand this way, a system must:

i. be XPG3 Base or XPG4 Base branded

ii. be based on USL/Novell code

iii. pass the System V Verification Suite (SVVS)

iv. commit to full XPG4 UNIX branding as quickly as possible.

This UNIX 93 interim profile is not a classic profile, and is only described in the ''Standards of Quality'' section of the Trade Mark Licence Agreement. It is only considered to be an interim measure similar to XPG4 Commands and Utilities (1992), allowing the marketplace to catch up to the specification programme.

Chapter 2

General Portability Issues

This chapter explains differences between Issue 3 and Issue 4 in the following areas:

- definitions of terms
- internationalisation and its effect on character set, locale, environment variables and regular expressions
- directory structure and devices
- general terminal interface
- interactive considerations.

2.1 Glossary

In Issue 3, each of the Base specifications (Commands and Utilities, System Interfaces and Headers) has its own chapter on definitions. In Issue 4, common definitions are contained in a single volume, the XBD specification.

The Glossary is substantially rewritten in Issue 4, containing a formal definition of more terms than are presented in Issue 3; the definitions themselves are more concise.

2.2 Internationalisation

Many of the features for internationalising applications in the POSIX-2 standard were developed by X/Open members and are related to features in Issue 3. Therefore, applications complying with Issue 3 have comparatively fewer migration difficulties than applications on other systems. A more complete description of the various internationalisation features can be found in the X/Open Internationalisation Guide, Version 2.

The major areas affected by internationalisation are the character set (see Section 2.3 on page 15), locale (see Section 2.4 on page 18), environment variables (see Section 2.5 on page 23) and regular expressions (see Section 2.6 on page 26).

2.2.1 Bytes and Characters

Many interfaces were originally designed with the assumption that a character consists of one byte. The interfaces in Issue 4 do not make that assumption. The XSI specification set makes precise use of the terms *byte*, *column position*, *octet* and *character*.

A byte is a unit of eight bits or larger. Sizes greater than eight are relatively uncommon, but nine, ten and even 16-bit bytes are possible. A byte exactly eight bits long is referred to as an octet, but octet refers to a quantity of eight bits in a row, not necessarily at an addressable machine boundary, as a byte is.

A character is defined as a sequence of one or more bytes representing a single graphic symbol or control code. Thus, wherever the term is used in the XSI specification set, the implication is that the referenced data object can contain multi-byte sequences.

Emerging encoding standards have defined characters of up to four 8-bit bytes. There is no requirement that every character is the same length, even within a string or file.

A column position has a width based on the narrowest displayable character in the character set. Although most western characters are all one column position wide, some character sets, such as Kanji, have double- or triple-width characters. Utilities such as *expand*, *fold*, *pr* and *vi* are aware of these relative widths. The width of each character is a value related to the current codeset in the locale. This information is not available to the application in a portable manner.

2.2.2 Message Catalogues

A number of schemes have been developed to handle text string data in applications programs, such that the appropriate string is delivered by the program in the appropriate language depending upon a run-time configuration. Tools to collect and manage these strings as ‘‘message’’ files or message catalogues, along with the programmatic interfaces to retrieve the strings at run time, exist in a number of forms on traditional UNIX systems.

Issue 4 provides the same interfaces as Issue 3, namely *catopen*( ), *catgets*( ), *catclose*( ) and the *gencat* utility. Programs that have made use of other schemes will need to migrate to these interfaces for portability across X/Open XPG4 UNIX and XPG4 Base branded systems.

2.3 Character Set

A portable filename character set is defined in Issue 3 for transferring the names of files between systems, but the only reference to a ‘‘portable character set’’ for text is made in an oblique reference to the 7-bit ASCII codeset, which is defined to be present in all supported locales.

By comparison, Issue 4 fully defines a portable character set in terms of the set of characters that are guaranteed to be included in all supported locales. Although no specific encoding is defined for the portable character set, certain rules are stated for encoding, as follows:

- A null character, NUL, must be present with all bits set to zero.
- For the encoding of characters 0 to 9, the value of each character after 0 must be one greater than its predecessor.
- All characters must be single-byte entities representable as a positive value of type **char**.

No guarantees are given or implied that the encoding of characters defined in the portable character set is invariant across locales. On systems where the encoding does vary, the results of applications accessing these locales are unspecified.

Another important difference between Issue 3 and Issue 4 is that the latter provides for the support of multi-byte codesets, which may or may not have state dependencies. Thus, for example, a text sequence may contain single-byte characters, characters that change shift state, or multi-byte characters. There is no requirement that all XSI-conformant systems support this feature, but XSI permits it.

2.3.1 Character Set Description File

This is a new feature in Issue 4. The provision of character set description files allows character sets and their mapping to coded character sets to be defined in a portable manner. In essence, a character set description file contains the list of symbolic characters defined in a codeset and their encodings. These are referred to throughout XPG4 as *charmap* files and are of particular significance to the *localedef* utility defined in the XCU specification.

2.3.2 Transparency

In Issue 4, all the utilities are transparent to 8-bit data. This includes all command-line arguments and file or character data processed by a utility. This should impose no migration barriers.

However, there are still portability restrictions on some 8-bit data when transported between systems:

- Inter-system mail may be restricted to 7-bit data by underlying mail or network protocols.
- 8-bit data and filenames may or may not be portable to systems that are not XSI-conformant.
- Filenames should be limited to those in the portable filename character set for full portability to POSIX systems.
- New restrictions in Issue 4 recommend that only characters in the portable character set in the XBD specification with the specific 7-bit ISO/IEC 646:1991 standard International Reference Version encoding be used for fully portable interchange of data.

2.3.3 Utilities and Multi-byte Codeset Support

It is important for the utility to describe exactly what units it is manipulating. Most of the utilities in Issue 4 deal with characters, although some, such as *fold*, have options to recognise either byte or character boundaries, and some, such as *wc* (using the **–c** option), deal only with bytes. Applications that were written with 8-bit characters in mind should be re-evaluated to find unportable assumptions. For example, if a file is translated to another codeset, the results of *wc* may differ. File sizes and all system limits ({PATH_MAX}, {LINE_MAX}, and so on) are generally expressed in bytes, which may be confusing to users who use codesets that encode characters with multiple bytes.

There are some restrictions imposed on implementations that use bytes larger than octets:

- Utilities that accept byte input in terms of numeric values, such as *awk* and *tr*, and those that can display bytes numerically, such as *ed*, *ex*, *od*, *sed* and *vi*, have no portable method of representing 9-bit bytes in hexadecimal or 10-bit bytes in octal.
- The *cksum* and *uuencode* utilities operate in terms of octets and portable results are not guaranteed for systems where the number of bits in a byte is not evenly divisible by 8.

2.3.4 System Interfaces and Headers Multi-byte Codeset Support

Many of the system interfaces and headers published in earlier versions of the XSH specification are defined to work exclusively with byte values, which limits them in terms of multi-byte codeset support. Ideally, these interfaces should be redefined to work with wide-character types. However, for reasons of compatibility with earlier systems, this approach is untenable.

This problem has been solved by defining a parallel set of interfaces. These are named collectively by X/Open as the Worldwide Portability Interfaces (WPI). They are identified in the XSH specification by WP markings in the left margin. The majority are aligned with the MSE working draft; exceptions are documented in Chapter 7. (The only exceptions are *wcsftime*( ), *wcstok*( ) and *wcswcs*( ).) These interfaces are functionally equivalent to the standard interfaces, but work on wide-character values. The single-byte versions are retained in the XSI for compatibility with Issue 3, but it is the WPI interfaces that are recommended for use by character-based portable applications. However, application developers should note that because these functions are aligned with the MSE working draft, the interfaces may change in the next issue of the XSH specification.

Externally, multi-byte sequences tend to be represented as strings of bytes (that is, type **char*** in C). As an object of this type would be difficult to manipulate internally, a new data type is introduced (**wchar_t**), which is defined as an object large enough to hold the binary encoded value of any character in the range of codesets supported by an implementation. The size of **wchar_t** can vary from one system to another, and functions are provided to convert from the external **char** format to the internal **wchar_t** format.

An implementation's maximum number of bytes in a character is defined by the constant {MB_LEN_MAX}; the maximum allowable number of characters in the current locale is indicated by {MB_CUR_MAX}.

Use of the WPI interfaces saves the programmer from having to worry about the differences between external and internal formats, as any conversions required are normally performed automatically when data is moved between different storage media. All the WPI interfaces manipulate character strings in terms of **wchar_t *** objects, allowing an implementation to support either single-byte or multi-byte codesets.

Note: All XSI-conformant systems are required to support the Worldwide Portability Interfaces, but whether or not they also support multi-byte codesets is implementation-dependent. Thus, it is permissible for a system supporting only single-byte codesets to define **wchar_t** as **char**.

2.4 Locale

Issue 4 contains a complete and coherent definition of the program locale, as a generic object, and a specific description of the POSIX (or C) locale supported on all XSI-conformant systems.

The Issue 4 locale definition is compatible with Issue 3, with the following additions:

- LC_MESSAGES is added to the list of locale categories.
- The rules for initialising a program locale from the environment are modified to cater for the *LC_ALL* environment variable (see Section 2.5 on page 23 for information on environment variables).
- Collation rules are extended to cater for multiple pass collation algorithms (rather than the simple two pass algorithms supported in Issue 3).

With respect to the definition of the POSIX locale, the following changes should be noted:

- Collation ordering is defined by the use of symbolic names rather than character codes. It is further mandated that the collation order is the numeric ordering of their ASCII codes, although it is not required that the character encoding is 7-bit ASCII.
- For category LC_CTYPE, a new keyword, **blank**, is added, which contains both spaces and tabs.
- The LC_MESSAGES category, and the **langinfo** constants NOEXPR and YESEXPR, are added. NOSTR and YESSTR are retained for compatibility with Issue 3.
- For category LC_MONETARY, the settings of values in the **lconv** structure are defined. In Issue 3, only the **langinfo** item CRNCYSTR is defined in this category.
- For category LC_NUMERIC, the settings of values in the **lconv** structure are defined.
- Items T_FMT_AMPM, ERA, ERA_D_FMT and ALT_DIGITS are added to the LC_TIME category.

The *localedef* utility is new in Issue 4; this gives applications and users the ability to create private locales by customising locale categories.

In general, although applications have had to be aware of locale states in previous issues, the advent of private locales is cause for much greater caution in the design of portable applications.

There are various categories within the overall locale; application developers should not modify all the categories, using LC_ALL, when only certain category effects are required.

Rules can be defined for each supported locale category as follows:

- LC_CTYPE defines rules for character classification and case conversion.
- LC_COLLATE defines rules for collation.
- LC_MONETARY defines rules for the formatting of monetary numeric information.
- LC_NUMERIC defines rules for the formatting of non-monetary numeric information.
- LC_TIME defines rules for the formatting of date and time information.
- LC_MESSAGES defines the format and values for affirmative and negative responses, and may also affect the format of messages from utilities.

Private locales are presented in what are generically known as *localedef* source files, which are processed by the *localedef* utility to produce a locale object from a portable source description of a locale. These locales do not affect the system in general or other users, but they do affect the operations of many of the standard utilities and some of the C-language interfaces.

The ramifications of this are discussed in the following sections.

2.4.1 Collation

Since users can arbitrarily modify the collation sequence with *localedef*, applications cannot rely on the sequence of output of such utilities as *ls* or *sort*, or those that use regular expressions or pattern matching, unless the application switches to a known locale for collation. For example, if the user has altered the collation of xaa, xab, and so on, an application that relies on a sequence such as:

```
split myfile
...
cat x?? > myfile
```

may get a scrambled file. This is because *split* creates files in a specific sequence, unaffected by the LC_COLLATE category, but pathname expansion is affected by it. In this example, replace the *cat* command with:

```
Save_COLLATE=$LC_COLLATE
LC_COLLATE=POSIX
cat x?? > myfile
LC_COLLATE=$Save_COLLATE
```

In the XCU specification, the following utilities are affected by LC_COLLATE:

Utilities Affected by LC_COLLATE				
awk	*cxref*	*join*	*nm*	*sort*
cflow	*dircmp*	*lex*	*pax*	*tar*
comm	*ed*	*localedef*	*pg*	*tr*
cp	*ex*	*ls*	*rm*	*uucp*
cpio	*expr*	*more*	*sed*	*vi*
csplit	*find*	*mv*	*sh*	*xargs*
ctags	*grep*	*nl*		

In the XSH specification, the following functions are affected by LC_COLLATE:

Functions Affected by LC_COLLATE				
fnmatch( )	*nl_langinfo*( )	*regexec*( )	*strxfrm*( )	*wcsxfrm*( )
glob( )	*regcomp*( )	*strcoll*( )	*wcscoll*( )	*wordexp*( )
localeconv( )				

2.4.2 Character Classification

Changing LC_CTYPE can be more invasive than LC_COLLATE because, in addition to regular expressions and shell patterns, character classification affects command-line processing, recognition of white space, display of non-printable characters by the editors, default field separators in *sort*, and so on.

In the XCU specification, the following utilities are affected by the LC_CTYPE category:

Utilities Affected by LC_CTYPE				
admin	*cut*	*join*	*pcat*	*tr*
alias	*cxref*	*kill*	*pg*	*tsort*
ar	*date*	*lex*	*pr*	*tty*
asa	*dd*	*lint*	*printf*	*type*
at	*delta*	*ln*	*prs*	*ulimit*
awk	*df*	*locale*	*ps*	*umask*
basename	*diff*	*localedef*	*read*	*unalias*
batch	*dircmp*	*logger*	*renice*	*uname*
bc	*dirname*	*logname*	*rm*	*uncompress*
bg	*dis*	*lp*	*rmdel*	*unexpand*
c89	*du*	*lpstat*	*rmdir*	*unget*
cal	*echo*	*ls*	*sact*	*uniq*
calendar	*ed*	*m4*	*sccs*	*unpack*
cancel	*env*	*mail*	*sed*	*uucp*
cat	*ex*	*mailx*	*sh*	*uudecode*
cc	*expand*	*make*	*sleep*	*uuencode*
cd	*expr*	*man*	*sort*	*uulog*
cflow	*fc*	*mesg*	*spell*	*uuname*
chgrp	*fg*	*mkdir*	*split*	*uupick*
chmod	*file*	*mkfifo*	*strings*	*uustat*
chown	*find*	*more*	*strip*	*uuto*
cksum	*fold*	*mv*	*stty*	*uux*
cmp	*fort77*	*newgrp*	*sum*	*val*
col	*gencat*	*nice*	*tabs*	*vi*
comm	*get*	*nl*	*tail*	*wait*
command	*getconf*	*nm*	*talk*	*wc*
compress	*getopts*	*nohup*	*tar*	*what*
cp	*grep*	*od*	*tee*	*who*
cpio	*hash*	*pack*	*test*	*write*
crontab	*head*	*paste*	*time*	*xargs*
csplit	*iconv*	*patch*	*touch*	*yacc*
ctags	*id*	*pathchk*	*tput*	*zcat*
cu	*jobs*	*pax*		

In the XSH specification, the following functions are affected by the LC_CTYPE category:

Functions Affected by LC_CTYPE				
atof( )	*ispunct*( )	*iswpunct*( )	*setlocale*( )	*towlower*( )
atoi( )	*isspace*( )	*iswspace*( )	*scanf*( )	*towupper*( )
atol( )	*isupper*( )	*iswupper*( )	*sprintf*( )	*vprintf*( )
fprintf( )	*iswalnum*( )	*iswxdigit*( )	*sscanf*( )	*wcstod*( )
fscanf( )	*iswalpha*( )	*mblen*( )	*strftime*( )	*wcstol*( )
isalnum( )	*iswcntrl*( )	*mbstowcs*( )	*strptime*( )	*wcstombs*( )
isalpha( )	*iswctype*( )	*mbtowc*( )	*strtod*( )	*wcstoul*( )
iscntrl( )	*iswdigit*( )	*nl_langinfo*( )	*strtol*( )	*wctomb*( )
isgraph( )	*iswgraph*( )	*printf*( )	*strtoul*( )	*wctype*( )
islower( )	*iswlower*( )	*regcomp*( )	*tolower*( )	
isprint( )	*iswprint*( )	*regexec*( )	*toupper*( )	

2.4.3 Date and Time Formatting

In Issue 3, the LC_TIME category affects utilities such as *date*, *ls* and *pr* only on certain internationalised systems. In Issue 4 all systems support this effect.

In the XCU specification, the following utilities are affected by the LC_TIME category:

Utilities Affected by LC_TIME				
ar	*cpio*	*ls*	*pr*	*uustat*
at	*date*	*mail*	*ps*	*who*
batch	*diff*	*mailx*	*tar*	
cal	*lp*	*patch*	*uucp*	
calendar	*lpstat*	*pax*	*uulog*	

In the XSH specification, the following functions are affected by the LC_TIME category:

Functions Affected by LC_TIME				
nl_langinfo( )	*setlocale*( )	*strftime*( )	*strptime*( )	*wcsftime*( )

2.4.4 Numeric Formatting

In Issue 3, the LC_NUMERIC category affects only the *sort* utility on some systems. The list is now expanded.

In the XCU specification, the following utilities are affected by the LC_NUMERIC category:

Utilities Affected by LC_NUMERIC				
awk	*od*	*printf*	*sort*	*time*

In the XSH specification, the following functions are affected by the LC_NUMERIC category:

Functions Affected by LC_NUMERIC				
fprintf( )	*nl_langinfo*( )	*setlocale*( )	*strfmon*( )	*wcstod*( )
fscanf( )	*printf*( )	*sprintf*( )	*strtod*( )	
localeconv( )	*scanf*( )	*sscanf*( )	*vprintf*( )	

2.4.5 Affirmative Responses

The value of the LC_MESSAGES category now affects the affirmative response characteristics of a number of utilities. Although this is primarily of concern to interactive users, application documentation may need changing. Programs such as the historical *yes* (not in the XCU specification) would have to be modified because the letter y is no longer a fully portable means of indicating agreement.

In the XCU specification, the following utilities have affirmative or negative response processing that is affected by the LC_MESSAGES category:

Utilities Affected by LC_MESSAGES Affirmative Responses				
cp	*find*	*mv*	*rm*	*xargs*
ex	*localedef*	*pax*	*tar*	

2.4.6 Message Formatting

In the XSH specification, the following functions are affected by the LC_MESSAGES category, although not for the purposes of determining affirmative or negative responses:

Functions Affected by LC_MESSAGES			
catopen( )	*nl_langinfo*( )	*perror*( )	*strerror*( )

2.4.7 Monetary Formatting

In the XSH specification, the following functions are affected by the LC_MONETARY category:

Functions Affected by LC_MONETARY			
localeconv( )	*nl_langinfo*( )	*setlocale*( )	*strfmon*( )

2.4.8 Functions that Invoke Commands

The *popen*( ) and *system*( ) functions are indirectly affected by all locale categories, because they invoke commands that may be affected.

2.5 Environment Variables

Applications should not create their own environment variable names without at least one lower-case letter. Collisions with future implementation extensions may otherwise result.

The definition of the *PATH* variable has changed. A portable application should only use . to represent the current directory in path searching. For example, the first *PATH* should be changed to the second:

```
PATH=:$HOME/bin::/bin:
PATH=.:$HOME/bin:.:/bin:.
```

The system still accepts both to mean the following sequence:

```
current directory
$HOME/bin
current directory
/bin
current directory
```

This is an artificial example to show the three possible locations of the current directory within *PATH*. It is not useful to specify the current directory more than once. Also, note that it is poor practice to precede the directories of system executable files with the current directory because it promotes a situation in which the security policy employed on the system is undermined.

Portable applications should not use hard-coded pathnames to access the standard utilities because the locations may vary between systems. Using the command:

```
getconf PATH
```

will provide a string with the names of the appropriate directories.

The concept of regulating language and cultural aspects of an application based on the contents of the *LANG* and *LC_** environment variables was introduced in Issue 2 and expanded upon in Issue 3. Thus, most XSI-conformant applications are already cognizant of these variables.

The following table provides a list of the environment variable names newly reserved in Issue 4. Applications that use any of these variables for other than the stated purpose should be modified:

Newly Reserved Environment Variables				
ARFLAGS	*GET*	*LFLAGS*	*OLDPWD*	*PS3*
CC	*GFLAGS*	*LINENO*	*OPTARG*	*PS4*
CFLAGS	*HISTFILE*	*MAKEFLAGS*	*OPTERR*	*PWD*
CHARSET	*HISTSIZE*	*MAKESHELL*	*OPTIND*	*RANDOM*
ENV	*LC_ALL*	*MANPATH*	*PPID*	*SECONDS*
FC	*LC_MESSAGES*	*MORE*	*PROCLANG*	*YACC*
FCEDIT	*LDFLAGS*	*NPROC*	*PROJECTDIR*	*YFLAGS*
FFLAGS	*LEX*			

With Issue 4, Version 2, two more environment variables have been reserved: *DATEMSK*, *MSGVERB*.

2.5.1 Locale Environment Variables

LC_ALL contains the user's requirements for the program's entire locale and overrides settings of the other locale environment variables (that is, *LC_COLLATE*, *LC_CTYPE*, *LC_MESSAGES*, *LC_MONETARY*, *LC_NUMERIC*, *LC_TIME* and *LANG*), when the *setlocale*( ) function initialises the program locale using the locale environment variables.

LC_MESSAGES affects the format of affirmative and negative responses, and also directs the *catopen*( ) function in determining the name of message catalogues, provided the value of the second argument in the call to *catopen*( ) is NL_CAT_LOCALE.

The significance of *LC_ALL* is that it fundamentally changes the rules by which a program locale is initialised from the environment. Thus, when the *setlocale*( ) function is called as follows:

```
setlocale(LC_ALL, "");
```

the locale is initialised according to which of the following conditions is satisfied first:

1. *LC_ALL* is set (this takes precedence over all other locale-related environment variables).
2. *LC_ALL* is not set and one or more of the category-specific environment variables is set.
3. *LANG* is set (this provides a default when neither *LC_ALL* nor a category-specific environment variable is set).

LC_MESSAGES replaces *LANG* in determining the language of affirmative and negative responses. It also replaces *LANG* in the affairs of the *catopen*( ) function if, and only if, the second argument is set to NL_CAT_LOCALE; otherwise the *catopen*( ) function still uses *LANG*, thus providing backwards compatibility with Issue 3. In particular, the description of the *NLSPATH* environment variable is changed to indicate use of *LC_MESSAGES* rather than *LANG* in determining the name and location of message catalogues.

Note: Guidelines for the location of message catalogues have not yet been developed. Therefore, take care to avoid conflicting with catalogues used by other applications and the standard utilities.

The format of locale names, as assigned to one of the locale-related environment variables, has also been changed from:

```
language[_territory[.codeset]][@modifier]
```

to:

```
language[_territory][.codeset][@modifier]
```

Thus, whereas the **territory** and **codeset** elements of a locale name are additive in Issue 3, they are independent in Issue 4.

Another change in Issue 4 is that if the value of a locale name begins with a slash, it is interpreted as the pathname of a locale definition.

2.5.2 Other Environment Variables

An alternative format is defined for the *TZ* environment variable:

```
:characters
```

which defines that the value is handled in an implementation-dependent manner.

In Issue 3, *TMPDIR* is mentioned as a variable that is commonly defined by implementations, but its use is not defined and its availability is not guaranteed.

2.6 Regular Expressions

The full syntax of regular expressions is consolidated in the XBD specification. In Issue 3, these definitions were split between two volumes (Commands and Utilities and Supplementary Definitions). Also, a statement is added indicating that the definition of historical simple regular expressions will be withdrawn in a future issue in favour of the improved internationalised version.

The definition of internationalised regular expressions is the same as in Issue 3, but they are now part of the POSIX.2 C-language Binding feature group. If they are supported, the feature test macro {_POSIX2_C_VERSION} is set to a value other than –1.

Where applications use regular expressions, it is recommended to use *regcomp*( ), *regexec*( ), and so on, rather than the *regexp*( ) family of functions, the *re_comp*( ) family of functions, or the *regcmp*( ) family of functions, all of which are marked TO BE WITHDRAWN. The latter two sets of functions were introduced with the X/Open UNIX Extension to support historical use.

Only rare compliant Issue 3 applications are impacted negatively by regular expression syntax changes in Issue 4 because X/Open was involved in their development. Existing applications should already be aware that the bracket expressions beginning with [[:, [[= and [[. are reserved for international character classes, equivalence classes and collating symbols respectively. It is possible that these obscure sequences were in use in some applications, because they were valid previously, but no portable Issue 3 application would use them.

The following changes may affect applications:

- A fully portable application should not use range expressions. They continue to be supported, but their use is discouraged. Applications that switch to a known locale for collation can use range expressions reliably, but that type of application has limited international usefulness. The vast majority of existing range expressions can be replaced by lists of specific characters or character classifications. For example:

Convert from	Convert to
`[a-z]`	`[[:lower:]]`
`[a-zA-Z]`	`[[:alpha:]]`
`[a-zA-Z0-9]`	`[[:alnum:]]`

- Range expressions can match multi-character collating elements. For example, if the multi-character collating element **ch** is defined and it collates between c and d, the expression:

  ```
  [c-d]at
  ```

  would now match the string:

  ```
  chat
  ```

  as well as:

  ```
  cat
  ```

  as it did previously.

- The Basic Regular Expression (BRE) interval expression syntax:

  ```
  \{m,n\}
  ```

  is extended to apply to all regular expressions. Previously, only *ed* and *sed* used this form.

CW For *grep* –**E**, *egrep*, *lex* and affirmative or negative responses (Extended Regular Expressions), the syntax is the same, less the backslashes. Previously, this matched actual characters.

For example:

```
a{1}
```

matched that literal four-character string. Such uses of braces surrounding numbers must be changed to escape the braces:

```
a\{1\}
```

- Regular expressions support both duplication and interval expressions following sub-expressions and back-references. Some historical systems supported:

  ```
  \n*
  ```

  but treated:

  ```
  \n\{min,max\}
  ```

  or:

  ```
  \( . . . \)*
  ```

  or:

  ```
  \( . . . \)\{min,max\}
  ```

  as invalid.

CW

- An implementation may treat multiple duplication symbols as an error. Previously, they were additive. As an example:

  ```
  a+*b
  ```

  matched zero or more instances of a followed by b. Now, multiple duplication symbols are undefined; that is, they cannot be relied upon for portable applications. In BREs, adjacent instances of * and:

  ```
  \{mn\}
  ```

  must be recoded. In EREs, adjacent instances of *, ?, + and {*m*,*n*} must be recoded.

CW

- An implementation may treat a circumflex or dollar-sign as an anchor within BRE sub-expressions. For example:

  ```
  \(^a\)b
  ```

  previously matched the string ^ab. Now, some systems may treat the circumflex as an anchor, matching ab at the beginning of a line. Any instances of:

  ```
  \(^
  ```

  or:

  ```
  $\)
  ```

  in BREs must be changed to escape the ^ and $.

  This change does not apply to EREs; these characters always required escaping to represent themselves outside bracket expressions.

- There is now a limit of at least 256 bytes on regular expressions. This is considerably larger than most historical systems and allows more elaborate patterns, particularly in *egrep*.

Utilities Employing Basic Regular Expressions				
csplit	*ex*	*more*	*pg*	*sed*
ctags	*expr*	*nl*	*red*	*vi*
ed	*grep*	*pax*		

Utilities Employing Extended Regular Expressions				
awk	*ex*[1]	*lex*[2]	*pax*[1]	*tar*[1]
cp[1]	*find*[1]	*mv*[1]	*rm*[1]	*xargs*[1]
egrep	*grep* **-E**			

Notes:

1. Used only as part of affirmative or negative response processing.
2. There are significant differences between the *lex* EREs and those described in the XBD specification, Section 7.4, Extended Regular Expressions.

Pattern Matching

Shell pattern matching adopts all of the internationalisation attributes of regular expressions. Further information on filename matching can be found in Chapter 4.

2.7 Directory Structure and Devices

2.7.1 Directory Structure

Issue 4 states that the following directories must exist on all XSI-conformant systems:

- /
- /dev
- /tmp

While it is generally safe to assume that these always existed on any XSI-conformant system, it is not stated explicitly in previous issues.

2.7.2 Output Devices and Terminal Types

The XBD specification, Section 8.2, Output Devices and Terminal Types, is added to indicate how implementations should support utilities defined in the XCU specification that have requirements for particular terminal characteristics. Note the following:

- Implementations are not required to support all terminal types. In particular, with asynchronous and synchronous devices, neither or both may be supported. In the case of asynchronous devices, they may support some or all of the required terminal characteristics.
- When a feature or utility is not supported on a specific terminal type, as allowed by the definition in the XCU specification, an implementation is required to indicate such conditions through diagnostic messages or exit status values (or both).
- An implementation is required to document which terminal types it supports and which features or utilities are not supported by each terminal.

2.8 General Terminal Interface

In Issue 3, job control is defined as optional; in Issue 4, it is mandatory. Thus, a number of changes are made to the General Terminal Interface, removing caveats like ‘‘If the implementation supports job control’’. This change also means that the signals SIGTSTP, SIGTTIN and SIGTTOU must be supported by an implementation.

For FIPS alignment, {_POSIX_VDISABLE} is supported on all XSI-conformant systems in Issue 4, meaning that all changeable special control characters can be disabled individually. This is defined as optional in Issue 3.

In the list of input modes flags, IUCLC is marked TO BE WITHDRAWN. OLCUC is similarly marked in the list of output mode flags, and XCASE is so marked in the list of local mode flags.

2.9 Invoking Commands

2.9.1 Utility Syntax Guidelines

Many of the utilities in Issue 4 are modified slightly to improve the consistency of command-line argument handling. This effort has had a much larger impact on implementations than on users, essentially requiring that almost all the standard utilities be recoded using *getopt*( ) and that backward compatibility for existing applications be maintained. The following changes have ramifications on applications and users:

CW

- Virtually all arguments of the form **–x***arg* are changed so that both **–x***arg* and **–x** *arg* are accepted on the command line. A portable application must use the **–x** *arg* form, splitting the option name and its option-argument into separate command-line arguments.

  There are a few exceptions to this policy. The POSIX-2 standard *pr* utility maintains the old form for historical reasons, and a portable application cannot rely on implementations supporting a new form (although they are allowed to do so). Also, some of the obsolescent forms maintain the old syntax.

  In the XCU specification, the following Issue 3 utilities have option-arguments that were previously adjacent to the option letter, but should now be separated:

Utilities Affected by Separating Arguments				
admin	*cxref*	*lp*	*prs*	*uucp*
awk	*delta*	*lpstat*	*ps*	*uupick*
cc	*dis*	*m4*	*rmdel*	*uustat*
cflow	*ex*	*nl*	*sort*	*val*
csplit	*get*	*paste*	*tabs*	*vi*
cut	*lint*	*pr*	*unget*	*xargs*

- A number of the utilities are modified to add options to replace forms of syntax that did not meet the Utility Syntax Guidelines:

  — single hyphens used as other than filename operands (for standard input or output)

  — option letters that are actually decimal values.

  The historical forms of these utilities are preserved as obsolescent versions. Portable applications should migrate to the new forms; details appear in Chapter 4. The following utilities are affected:

Utilities Modified by the Syntax Guidelines				
du	*fgrep*	*lex*	*sort*	*tty*
ed	*get*	*more*	*split*	*uniq*
egrep	*head*	*newgrp*	*strings*	*uux*
env	*join*	*nice*	*tail*	*vi*
ex	*kill*	*nl*	*touch*	*xargs*
expand				

- Most of the utilities are now required to accept −− as an operand that means ‘‘no more options follow’’. Portable applications should be modified to use this argument whenever operands are being constructed for a command line and the application writer does not have full control over the form of the operand. For example, any operand that is a pathname, a database entry or a user reply could be a problem.

The script:

```
printf "Enter filename(s) to be removed:"
read Reply
eval rm $Reply
```

is not portably consistent if there is a chance the first filename entered by a naive user could be:

```
-rf
```

This is admittedly not a portable filename, but it is valid. In this example, the utility name *rm* should be followed by the −− argument. However, if the application writer is supplying the operands as part of the script itself, this change is unnecessary, although benign.

This guideline also applies to utilities that do not usually take any options because the implementation is free to add options as extensions.

2.9.2 Limits

Various limits now affect the behaviour of utilities. In all cases, these limits are in excess of the limits imposed by historical systems (which frequently differed between systems and were rarely publicised).

- The {LINE_MAX} limit is the most significant change. In most cases, this change does not affect existing applications. Many of the standard utilities previously had limits on processing text lines in the range of 128 to 512 bytes; some would fail in unpredictable ways if the line length were exceeded. Therefore, portable applications in the past avoided long lines. New applications are assured of successful processing for lines no longer than 2048 bytes.

  The application should be able to handle such long lines. For example, a C-language application that receives pipeline input from a standard utility must now be aware that files with long lines may appear on systems, and the utilities now pass them through. In C, the current value of {LINE_MAX} is available in **<limits.h>** or from *sysconf*( ), but it is never less than 2048.

  Shell script applications should be evaluated for the inclusion of *fold*, if displaying, printing or mailing longer lines is anticipated.

- Some utilities now have specific limits that increase their capacity over historical versions and make using them more consistent and portable. All those processing regular expressions can now handle arbitrary expressions up to 256 bytes. The *sed* stream editor can process larger pattern and hold spaces. Other examples of increased limits are in Chapter 4.

- A related fact is that many previous arbitrary limits have been silently removed. For example, previous versions of *ed* were limited to comparatively small editing buffers. The new version of *ed* has no stated editing size limit. Therefore, the implementation dynamically allocates resources to provide what is required. If *ed* fails, it is now because some system resource (for example, memory, disk space, number of active processes) was exhausted, not because of an arbitrary programming limit, such as a fixed size internal buffer. Also, the commands in *sed* are no longer arbitrarily limited.

  Another example of a removed, unstated limit is that some versions of *rm* would fail to remove a directory hierarchy if it contained too many levels of sub-directories.

2.9.3 Input/Output Formats

Many informative (as opposed to diagnostic) messages from the utilities are documented, to promote their use within pipelines in a portable way. Applications should be examined for any previous reliance on specific message contents. In most cases, these messages were not listed in Issue 3, so they could not have been used portably.

The new message specifications are written such that usage of white space can be varied by the utility, to line up columns, for example. Applications cannot generally rely on comparisons of fixed strings, but should be written to examine the utility output with *awk*, *lex*, *read* or *scanf*( ).

Applications relying on the new message formats must be aware of the locale in which they are operating. In most cases, the XCU specification specifies formats only when *LC_MESSAGES* is set to the POSIX locale.

Unfortunately, it is still difficult to parse outputs from some utilities in a portable way. For example, if a utility message includes a filename, that name could potentially include space, tab or newline characters, making it difficult to locate field, or even line, boundaries. Applications making heavy use of output formats are probably portable only to systems with conscientious users and system administrators who follow portable filename guidelines.

2.9.4 Errors

Exit status values are listed for the first time for many utilities, although it is frequently the case that this is ‘‘zero success, non-zero failure’’. In those cases, applications should be examined and any reliance on system-specific exit values removed. For example, relying on *pack* to return the number of files it could not pack was never portable in cases where that number could have been 256.

Some exit status values are reserved for specific uses:

126 A utility requested to be invoked was found but could not be invoked.

127 A utility requested to be invoked could not be found.

>128 The utility was interrupted by a signal.

Writing applications that exit with these values for reasons other than those stated should be avoided. Utilities that list these exit values return them only in the cases specified. Other utilities may return these values for other conditions.

2.9.5 Other Global Utility Behaviour

The definition of a *text file* requires that each line is terminated with a newline. Some historical versions of utilities may have allowed the last newline to be missing, silently adding one in the output. This behaviour is not guaranteed, except where specifically documented. Applications creating text files should be examined to ensure they are doing so correctly.

Implementations are required to allow simultaneous execution of some utilities, even if historically they could not because of temporary-file naming conventions that caused collisions in common directories. They are also required to clean up temporary files in most cases.

XPG3-XPG4 Base Migration Guide, Version 2

Part 2:

Commands and Utilities Migration

X/Open Company Ltd.

Chapter 3

Shell Command Language

This chapter considers the effects of new features in the Issue 4 shell command language. In most cases these offer opportunities for new applications to be written with more reliance on the shell itself and less on the utilities. In some cases, however, these new features require subtle changes to existing applications.

3.1 Naming Considerations

3.1.1 Identifiers

The letters in portable *names* are restricted to those in the portable character set; this is not stated in Issue 3. *Alias names* can also include the characters:

```
!   %   ,   @
```

Implementations supporting additional characters document whether those alphabetics can be used in names and aliases.

3.1.2 Operators

CW The symbol ((is reserved as a control operator on some systems. Therefore, nested sub-shells that begin with ((must be separated. For example, convert:

```
((echo hello);(echo world))
```

to:

```
( (echo hello);(echo world))
```

Some systems (for example, those using the KornShell to implement the XCU specification shell) may treat these as invalid arithmetic expressions instead of subshells. Note that this requirement does not force the separation of)) because the shell is able to distinguish the termination of arithmetic from that of nested subshells.

CW The ! character is now a reserved word that complements the results from a pipeline. This was previously a valid, although unportable, utility name.

The new redirection operators >| and <> are added, but these were not valid shell syntax previously.

The curly-brace { } characters are designated as possible control operators in a future issue; they are currently reserved words. Portable applications should begin quoting them now if they are to represent literal characters because this quoting may be required in the future. For example, (using one quoting mechanism), convert:

```
echo {Hi}
```

to:

```
echo \{Hi\}
```

CW The four words: **function**, **select**, **[[** and **]]** are reserved and cannot be used where a reserved word would be recognised, such as a command name.

3.1.3 Selecting Command Interpreters

CW Some systems have supported a kernel feature that caused special treatment of shell scripts beginning with the characters #!. This was used to select a command interpreter. For example, it is common to see a script beginning with:

```
#!/bin/sh
```

which means, ‘‘run this using the program **/bin/sh**, even if another shell is in use’’. This was never strictly portable (because the absolute pathname **/bin/sh** is not guaranteed on a system), and a fully portable program must not rely on it; the system may treat it only as a comment.

3.1.4 Aliases

Aliases are a new facility and should cause no forward compatibility problems.

However, if there is concern about the user setting up an environment where utility names do things unintended by the application, note that aliases can be brought into a shell script through common implementation extensions. A way to guard against command names expanding into aliases is to quote them. For example, a very common alias that an interactive user might set up is:

```
alias ls="ls -CF"
```

This would disrupt shell scripts such as:

```
ls | pax ...
```

The previous *ls* could be replaced by *l\s* or an *unalias ls* command could be issued.

3.1.5 Reserved Command Names

Because of new utilities and other changes, the following are no longer valid names for local commands, unless they are invoked with a pathname containing a slash:

- any unquoted command name ending with a colon (:)
- any of the following, when not quoted: !, [[or]]
- any of the following:

alias	fg	more	time
asa	fold	nice	tput
bg	fort77	pathchk	unalias
c89	function	pax	uncompress
cksum	head	printf	unexpand
command	jobs	renice	uudecode
compress	locale	sccs	uuencode
expand	localedef	select	zcat
fc	logger	strings	

3.2 Parameters, Variables and Word Expansions

3.2.1 IFS

The treatment of the value of the *IFS* variable and its use in field splitting are changed:

1. The *unset* special built-in can unset *IFS*. If not set, *IFS* behaves as if it were:

   ```
   <space><tab><newline>
   ```

CW 2. When "$*" is expanded, the first character of *IFS* is used as a separator. Although this is as stated in Issue 3, historical systems actually used a single space character in this instance.

CW 3. If *IFS* is null, there is no separator, for example:

   ```
   $ IFS=''
   $ set a b c
   $ echo "$*"
   abc
   ```

   Once again, previous systems used a space character.

CW 4. When *IFS* is set to a value other than:

   ```
   <space><tab><newline>
   ```

   the characters within *IFS* that are not white space act as field separators. For example, if a file **/etc/passwd** contained the first line:

   ```
   root::0:0::/:/bin/sh
   ```

   The script:

   ```
   IFS=":"
   read a b c < /etc/passwd
   echo $b
   ```

   would have previously produced 0, but it now produces a null value, indicating that each instance of a colon delimited a field.

5. Field splitting with *IFS* occurs only after parameter expansion, command substitution or as part of the *read* command. This prevents certain security loopholes. Previously, on some systems the following input:

   ```
   $ IFS=o
   $ violet
   ```

   would invoke *vi* to edit file **let**.

   No portable application should have relied on this behaviour.

3.2.2 Tilde Expansion

CW Previously, it was unportable but valid to have files named with a leading tilde character (~). Now, the use of such files in scripts should have quoting for the leading tildes, because the first component may match a login ID. For example, the first command should be converted to one of the following three:

```
cat ~jan ~feb ...
cat "~jan" "~feb" ...
cat \~jan \~feb ...
cat ./~jan ./~feb ...
```

Although tilde expansion was not used previously by portable applications, a common KornShell extension must be avoided:

```
PATH=~dwc/bin:~maw/bin
```

This does not expand the second tilde (because it does not start a word). Use one of the following instead:

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

```
PATH=$(printf %s ~dwc/bin : ~maw/bin)
```

3.2.3 Parameter Expansion

Five new forms of parameter expansion are added that yield string lengths and remove prefix or suffix patterns:

${*#parameter*}	String length.
${*parameter%word*}	Remove smallest suffix pattern.
${*parameter%%word*}	Remove largest suffix pattern.
${*parameter#word*}	Remove smallest prefix pattern.
${*parameter##word*}	Remove largest prefix pattern.

These can be used to replace some of the existing *expr* and *sed* calls in existing scripts and improve performance and readability in many cases.

The rules for parameter expansion with double-quotes:

```
"${...}"
```

are changed to require that any single- or double-quotes must be paired within the curly-braces. A consequence of this rule is that single-quotes cannot be used to quote the } within:

```
"${...}"
```

For example:

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

is invalid because the:

```
"${...}"
```

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

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

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

```
"‘echo hello"
```

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

```
sh -c ’‘ echo "foo‘‘
```

3.2.4 Command Substitution

Backquoted command substitution must be terminated by a backquote:

```
‘...‘
```

Some shells allowed the end of a file or string silently to delimit the command substitution.

A new form of command substitution is introduced that makes using quoting and nesting rules easier than with the back-quote method:

```
$(...)
```

It is unnecessary for any scripts to be converted to this new form, but new script writers may find it easier and more logical to use the new form for any complex constructs:

- When any of \$, \’ or \\ appear within back-quotes, the leading \ is removed:

  ```
  echo ‘echo ’\\$x‘‘
  ```

  as compared to:

  ```
  echo $(echo ’\$x’)
  ```

- Nesting back-quote command substitutions requires escaping the enclosed `‘...‘` as follows:

  ```
  echo ‘echo \‘echo Hi\‘‘
  ```

  as compared to:

  ```
  echo $(echo $(echo Hi))
  ```

If the new form is used to execute a subshell, care must be taken to remove any ambiguity arising from arithmetic expansion. For example, if a utility named: *1+2* is written, the command:

```
$((1+2))
```

is then ambiguous. It must be written portably as:

```
$( (1+2))
```

Single-quotes cannot be used to quote the } within:

```
"${...}"
```

3.2.5 Arithmetic Expansion

Arithmetic expansion is a new feature without forward compatibility problems. It can be used to simplify existing shell arithmetic that involves the *expr* utility.

3.3 Redirection

Multi-digit file descriptors are now allowed syntactically, although a portable application cannot use numbers higher than 9, because the shell can reserve all higher numbers for its own use.

The new *noclobber* version of redirection can be used for creating lock files in applications or determining that a file can be created safely without replacing an existing file. For example, consider an application that wishes to save a copy of a file before it edits it:

```
cat $1 > $1.back
```

On a traditional file system with 14-byte filenames, if $1 is ten bytes or larger, this command could erase another file. If $1 is 14 bytes, it would erase the original file instead of copying it. Now, the following can be written:

```
set -C    # set noclobber mode
if > $1.back; then
    echo Backup copy can be created
else
    echo Backup attempt will fail
```

The noclobber mode is even more useful for interactive users who wish to prevent inadvertent destruction of their files. They would then have to use the >| operator to overwrite a file deliberately.

The new <> operator has been supported on many implementations, but not documented. It should cause no compatibility problems, but its use is rather specialised.

CW Issue 3 allowed here-documents to be terminated by the end of the script file, as in the following example of a two-line file:

```
cat <<EOF
Hi
```

A fully portable script requires a proper delimiter.

The System V shell and the KornShell have differed historically on pathname expansion of an argument *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 capability was useful, but only as a shorthand device for interactive users. It is not reasonable to write a shell script such as:

```
cat foo > a*
```

Therefore this is not permitted.

3.4 Shell Commands

When a command names a utility that cannot be found, there is no assurance that this aborts a script. A portable script must be written to test the exit status of each command it considers critical before proceeding to the next step.

Historically, shells have returned an exit status of 128+*n*, where *n* represents the signal number. Since signal numbers are not standardised, 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. Therefore, a portable script cannot rely on determining the exact cause of a command failure when a signal is received.

There is a historical difference in *sh* and *ksh* non-interactive error behaviour. When a command named in a script is not found, some implementations of *sh* exit immediately, but *ksh* continues with the next command. Thus, the POSIX-2 standard says that the shell *may* exit in this case. This puts a small burden on the programmer, who has to test for successful completion following a command, when it is important that the next command not be executed if the previous is not found. When it is important for the command to be found, it is probably also important for it to complete successfully. The test for successful completion does not need to change.

With the System V shell, all built-ins are treated as special built-ins, which causes them to exhibit the special behaviour listed in the XCU specification, Section 2.14, Special Built-in Utilities (such as the difference in how variable assignments stay in effect). Earlier versions of System V and BSD systems did not implement the common *echo*, *pwd* and *test* as built-ins (regular or special), so these older systems are actually closer to the current XCU specification. The differences between the behaviour of built-ins and other utilities is not documented in the SVID.

3.4.1 Command Search

The rules for command search make explicit the differences between special and regular built-ins. Previously, regular built-ins had different characteristics from file-system utilities, but the portable application could not predict which utilities were which. So, it was impossible to write a shell function with the name of a common utility because that utility might be built-in and the function would never be accessed.

In Issue 4, an application cannot discern between regular built-ins and file-system utilities (unless it is able to check for performance differences). All utilities, other than the special built-ins, can be replaced with functions. All utilities, other than the special built-ins, can be used as if they were in the file system by commands such as:

```
nohup utility
find . -exec utility \;
ls * | xargs utility
```

It is important to understand that some utilities only affect or understand their own shell environment, not their parent's; commands such as:

```
(cd /tmp)
nohup kill %1
env wait
```

are valid, but not very useful.

The new *command* utility can be used to suppress function lookup.

3.4.2 Pipelines and Lists

A new reserved word, **!**, can be used to complement the exit status of a pipeline. For example:

```
if ! false; then
    echo True
fi
```

Scripts assuming that a pipeline is (or is not) executed in a subshell must be modified to tolerate the pipeline being executed in a subshell, but not to depend on it. For example, the command:

```
echo dog cat mouse | read x y z
echo $x $y $z
```

does not work as expected on some systems because the *read* is invoked in a subshell and does not affect the variables in the current environment. This example could be written as:

```
read x y z <<eof
dog cat mouse
eof
echo $x $y $z
```

Scripts written assuming that the first example using *read* would *not* work are also not portable, because some shells, such as the KornShell, do run the final pipeline stage in the current environment. Such a script could use:

```
echo dog cat mouse | (read x y z)
echo $x $y $z
```

to be sure *read* would not affect the current environment.

Portable applications should avoid using the AND and OR operators, && and | |, in complex constructs without using { } or ( ) groupings to show the precedence desired. The precedence of these operators, strictly left-to-right, is different from most programming languages, where AND has higher precedence, and confusion may result. So, for example, the following three commands are equivalent (assuming the subshell effects in the second are not relevant), but the second or third is preferred:

```
a || b && c
(a || b) && c
{ a || b; } && c
```

3.5 Pattern Matching

Pattern matching is expanded with the internationalisation features described for bracket expressions in Section 2.6 on page 26.

A period in a bracket expression may now match a leading dot in a filename. Previously this was never portable and a portable application still cannot use this form.

CW A leading circumflex in a bracket expression must be quoted. For example, to list filenames beginning with ^ or a, use one of the following:

```
ls [a^]*
ls [\^a]*
```

Any of the shell special characters used in a pattern must be quoted or escaped. In most cases, this was already necessary to prevent their misinterpretation by the shell. For example:

```
ls a(b*
```

never worked. However, the command:

```
find . -name 'a(b*' -print
```

did work.

CW Now, this form requires escaping of the shell character to avoid side effects from implementation extensions:

```
find . -name 'a\(b*' -print
```

3.6 Special Built-ins

Special built-ins have special properties for error conditions, variable assignments and accessibility via the *exec* functions and certain commands (such as *nohup*). It is now possible for the application to predict these effects because the list of special built-ins is specified; although any of the standard utilities could be implemented as a regular built-in, none of them can be special built-ins.

3.6.1 dot

Some older implementations searched the current directory for the file, even if the value of *PATH* disallowed it. This behaviour is now prohibited due to concerns about introducing susceptibility to trojan horses, which the user might be trying to avoid by leaving dot out of *PATH*.

3.6.2 exec

Most historical implementations were not conformant in that:

```
foo=bar exec cmd
```

did not pass **foo** to *cmd*. It is unlikely that any application ever relied on it not being passed.

CW Applications relying on file descriptors > 2 being automatically closed or left open following an *exec* must be recoded to force the desired result.

3.6.3 exit

In applications, the reserved exit status values 126 and 127 should be avoided, except as described in the XCU specification. Values greater than 128 should be reserved for signal terminations.

3.6.4 export

Instances without arguments that expect a specific portable output format must be recoded as:

```
export -p
```

Applications relying on previous output formats are not portable.

3.6.5 readonly

Instances without arguments that expect a specific portable output format have to be recoded as:

```
readonly -p
```

Applications relying on previous output formats are not portable.

3.6.6 return

The behaviour of *return* when not in a function or dot script differs between shells. In some shells this is an error; in others, and in Issue 4, the effect is the same as *exit*.

The exit value given to *exit* cannot exceed 255.

3.6.7 set

In some previous shells:

```
set --
```

only unset parameters if there was at least one argument; the only way to unset all parameters was to use *shift*. Using the new Issue 4 version should not affect existing scripts because there should be no reason deliberately to issue it without arguments; if it is issued as:

```
set -- "$@"
```

and there are in fact no arguments resulting from $@, unsetting the parameters would achieve the same effect.

The use of *set* + without other arguments (which is similar to *set* with no arguments, except that the values of the variables are not reported) is not documented in Issue 3 and is no longer supported. An application requiring this could substitute:

```
"set | sed"
```

to suppress the variable values.

CW The **–k** and **–t** options are no longer supported and should be removed from portable scripts.

3.6.8 trap

Applications should be migrated to the symbolic signal names.

Scripts relying on a specific system's *trap* output format have to be recoded.

3.6.9 unset

Applications must be recoded to use *unset* with either **–f** or **–v** to be fully portable.

Chapter 4

Utilities

4.1 Introduction

This chapter contains the migration information for the XSI utilities in the XCU specification. In general, most changes occurred between Issue 3 and Issue 4 of the XCU specification. Very little changed between Issue 4 and Issue 4, Version 2.

- All references to the ISO POSIX-2 DIS have been updated to the POSIX-2 standard, to reflect its proper current status as a full international standard.
- There were a number of minor corrections and clarifications made to the text of several utilities, namely: *awk*, *compress*, *make*, *od*, *pack*, *pcat*, *sact*, *uname*, *uncompress*, *unpack*.
- Functional additions were made to several utilities for conformance to X/Open UNIX Extension requirements, namely: *c89*, *cc*, *getconf*.

4.1.1 Symbolic Link Support

As shown in the XCU specification, Section 1.2.1, Symbolic Links, the definition of symbolic links and pathname resolution with respect to symbolic links in the XBD specification, Chapter 2, Glossary, is new for Version 2; however, support for symbolic links is not required to conform to the XCU specification. This document will be fully-aligned with the emerging IEEE Std 1003.2b (POSIX.2b) in a future edition, and it is that IEEE standard that fully specifies the behaviour of the utilities with respect to symbolic links. (Refer to the XCU specification, Section 1.6.1, Relationship to Emerging Formal Standards.)

4.2 Utility Migration Information

admin — create and administer SCCS files (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires separation by blank characters for: –**a** *login*, –**d** *flag*, –**e** *login*, –**f** *flag*, –**m** *mrlist*, –**r** *rel*, –**t** *name* and –**y** *comment*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

The *mrlist* option-argument is now required with all uses of –**m**. The *name* option-argument is now required with –**t** when –**i** or –**n** are used in creating a new SCCS file.

Input lines beginning with a character with a numeric value of 1 (SOH, or <control>-A) are no longer allowed. The various SCCS files are now required to be text files, but their formats are explicitly unspecified, disallowing universal inter-system portability of, for example, the **s.** files. Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats were never portable.

alias — define or display aliases

Issue 4: A new utility for Issue 4.

ar — create and maintain library archives

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

asa — interpret carriage-control characters

Issue 4: A new utility for Issue 4.

at — execute commands at a later time

Issue 4: Conformance to the Utility Syntax Guidelines requires a blank character between the option and argument for –**f** *file* and –**q** *queuename*.

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The spacing requirements for the *timespec* are now looser than some systems allowed.

awk — pattern scanning and processing language

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for –**F** *ERE*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

This description is based on the new *awk*, *nawk*, which introduced a number of new features to the version of *awk* in Issue 3:

1. New keywords: **delete**, **do**, **function** and **return**.
2. New built-in functions: *atan2*( ), *cos*( ), *sin*( ), *rand*( ), *srand*( ), *gsub*( ), *sub*( ), *match*( ), *close*( ) *and*( ) *system*( ).
3. New predefined variables: **FNR**, **ARGC**, **ARGV**, **RSTART**, **RLENGTH** and **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 match more closely C. Two examples of code that operate differently are:

```
while ( n /= 10 > 1) ...
if (!"wk" ~ /bwk/) ...
```

Several features are added for POSIX-2 standard alignment that are not documented in the referenced document by Aho, Kernighan and Weinberger.

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 ISO C standard.
6. Regular expressions are extended to make them a pure superset of Extended Regular Expressions (see the XBD specification, Section 7.4, Extended Regular Expressions).

The precedence of the **getline** function and the `|`, `<` or **concatenate** operators has formally been declared ambiguous. For example, a portable application cannot rely on:

```
getline < "a" "b"   or   getline < "x" + 1
```

always being parsed as either of the following:

```
(getline < "a") "b"   or   getline < ("x" + 1)
getline < ("a" "b")   or   (getline < "x") + 1
```

even though the first case in each pair is the most prevalent historically. Parentheses must be used to achieve the precedence desired.

Conversion between string and numeric values is slightly changed. In previous 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, in previous 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. Issue 4 ensures that comparisons are numeric if necessary. With previous 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. Issue 4 requires that the output of the second *print* statement is **`3.140000`**.

To avoid the problem of the following script printing nothing:

```
BEGIN {
    y[1.5] = 1
    OFMT = "%e"
    print y[1.5]
}
```

a new variable, **CONVFMT**, is 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.

Issue 4, Version 2:

Examples 10 and 17 in the **EXAMPLES** section have been corrected.

banner — make large letters (WITHDRAWN)

Issue 4: This utility is now withdrawn because its output could not be well specified in an internationalised environment (particularly for large, complex character sets), and it is rarely needed by portable applications. XSI-conformant systems continue to offer banner-like services for *lp* header pages.

basename — return non-directory portion of pathname

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The POSIX-1 standard 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. Historical implementations also differed in their handling of *suffix* when *suffix* matched the entire string left after removing the directory part of *string*.

The behaviour of *basename* and *dirname* in the XCU specification are coordinated so that when *string* is a valid pathname:

```
$(basename "string")
```

would be a valid filename for the file in the directory:

```
$(dirname "string")
```

batch — execute commands when the system load permits

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

bc — arbitrary-precision arithmetic language

Issue 4: A new utility for Issue 4.

bg — run jobs in the background

Issue 4: A new utility for Issue 4.

c89 — compile standard C programs

Issue 4: A new utility for Issue 4. See the entry for *cc*.

cal — print calendar

Issue 4: This utility is specified more fully than in Issue 3, but it should operate identically, except for the possible effects of the internationalisation variables on some systems.

calendar — reminder service (TO BE WITHDRAWN)

Issue 4: Input lines of calendar files are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters.

The *calendar* utility will be withdrawn from a future issue.

It would require considerable work to make it suitable for international use; emerging desktop productivity tools are expected to address this need better.

cancel — cancel line printer requests

Issue 4: This utility operates exactly as in Issue 3, except for the possible effects of the internationalisation variables on some systems.

cat — concatenate and print files

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The –**s** option (write no diagnostic messages when non-existent files are named) is marked PI in Issue 3, is not in the POSIX-2 standard, and differs in meaning on systems based on System V compared with systems based on BSD. Therefore, it is deleted from the XCU specification. An application needing to suppress messages (the System V meaning of –**s**) can use:

```
cat ... 2>/dev/null
```

An application needing to squeeze excess blank lines (the BSD meaning of –**s**) can use this script based on *sed*:

```
sed -n '
# Write non-empty lines.
/./   {
      p
      d
      }
# Write a single empty line, then look for more empty lines.
/^$/  p
# Get next line, discard the held newline character
# (empty line), and look for more empty lines.
:Empty
/^$/  {
      N
      s/\n//
      b Empty
      }
# Write the non-empty line before going back to search
# for the first in a set of empty lines.
      p
'
```

cc — a C-language compilation system (TO BE WITHDRAWN)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**e** *epsym*, –**D** *name***[**=*value***]**, –**I** *directory*, –**L** *directory*, –**o** *outfile*, –**u** *symname*, –**U** *name* and –**W** options. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

See Part 4 for additional information on C-language migration issues.

Issue 4, Version 2:

In the **Standard Libraries** subsection, the –**l c** operand describes access to traditional interfaces if _XOPEN_UNIX is defined by the implementation.

cd — change working directory

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

Applications relying on a directory named − can no longer do so. The *PWD* and *OLDPWD* variables are now modified by every successful invocation of *cd*.

cflow — generate C-language flowgraph (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**d** *num*, −**D** *num***[**=*def***]**, −**i** *incl*, −**I** *dir* and −**U** *dir*. The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

For all except binary input files, input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

chgrp — change file group ownership

Issue 4: Some systems count the number of errors and reflect that in the exit status code. Applications that rely on this behaviour are not portable.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

chmod — change file modes

Issue 4: Some systems count the number of errors and reflect that in the exit status code. Applications that rely on this behaviour are not portable.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *mode* operand beginning with −, such as −**r**, −**w**, −**s**, −**x** or −**X**).

Applications should be migrated to the symbolic permissions form.

chown — change file ownership

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Some systems count the number of errors and reflect that in the exit status code. Applications that rely on this behaviour are not portable.

chroot — change root directory for a command (WITHDRAWN)

Issue 4: This utility is withdrawn because there is no portable way to set up an environment where it is useful and it is usually usable only by applications with appropriate privileges.

cksum — write file checksums and sizes

Issue 4: A new utility for Issue 4.

For applications migrating from *sum* to *cksum*, the following changes apply:

- The checksum produced is completely different, but it is guaranteed to be calculated the same on all systems.
- The *sum* block count is changed to an octet count in *cksum*.

cmp — compare two files

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

col — filter reverse line-feeds (TO BE WITHDRAWN)

Issue 4: Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The *col* utility will be withdrawn from a future issue. It may be replaced in a future issue by a similar utility without the current ASCII bias.

comm — select or reject lines common to two files

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

command — execute a simple command

Issue 4: A new utility for Issue 4.

compress — compress data

Issue 4: A new utility for Issue 4.

Applications converting from the *pack* family must be cognizant of the different suffix used for the compressed files: upper-case **.Z**.

Issue 4, Version 2:
The **DESCRIPTION** section is clarified to state that the ownership, modes, access time and modification time of the original file are preserved if the invoking process has appropriate privileges. The **STDOUT** section includes the case where a *file* operand is –.

cp — copy files

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The −**i**, −**p**, −**r** and −**R** options are new in Issue 4.

When creating directories, some historical versions of *cp* used the mode of the source directory, plus read, write and search bits for the owner, as modified by the file mode creation mask. This was 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* fails. 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 behaviour is modified so that *cp* is always 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 behaviour is to prevent users from accidentally creating directories with permissions beyond those they would normally set and for consistency with the behaviour of *cp* in creating files.

cpio — copy file archives in and out (TO BE WITHDRAWN)

Issue 4: The capability of *cpio* is stabilised at the Issue 3 level. The *pax* utility is replacing *cpio* and is able to read and write archives produced by *cpio*.

crontab — schedule periodic background work

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The new −**e** option allows editing of the *crontab* entry.

csplit — split files based on context

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**f** *prefix* and −**n** *number*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines in the file to be split are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The new −**n** option allows files to be split into more than 99 files (the limit in previous issues).

ctags — create a tags file

Issue 4: A new utility for Issue 4.

cu — call another system

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

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

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**b** *list*, –**c** *list*, –**d** *delim* and –**f** *list*.

Input lines cannot contain NUL characters. Input files must end with a newline character.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

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 behaviour is prohibited by the XCU specification, where only the exit status is affected by this problem.

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. However, the *cut* –**c** option in the XCU specification treats neither backspace nor tab characters in any special fashion. The *fold* utility does treat these characters specially.

cxref — generate C-language program cross-reference table (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**o** *file*, –**w** *num*, –**D** *num***[**=*def***]**, –**I** *dir* and –**U** *dir*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

date — write the date and time

Issue 4: The following new field descriptors are added:

- %e (equivalent to the form used by *date*'s default output in the POSIX locale)
- %C (century)
- %u (weekday number, Monday as 1)
- %V (week number, starting at 1)
- modified field descriptors (%Ec, %Od, and so on).

The %U, %V and %W descriptors precisely define how week numbers are assigned in the first few days of the year.

Issue 3 incorrectly implies that GMT would be used if *TZ* is not set; this is currently implementation-dependent.

dd — convert and copy a file

Issue 4: Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

delta — make a delta (change) to an SCCS file (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**g** *list*, –**m** *mrlist*, –**r** *SID* and –**y** *comment*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

The *mrlist* option-argument is now required with all uses of –**m**.

The various SCCS files are now required to be text files, but their formats are explicitly unspecified, disallowing universal inter-system portability of, for example, the **s.** files.

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

df — report free disk space

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The new –**P** (portable) option produces output in a fully specified format, allowing the portable use of *df* in scripts. The –**t** option is retained as an extension, but it does not produce consistently formatted output on all systems. Shell script writers should use *df* –**P**.

Previous issues did not require that space figures be reported uniformly in 512-byte units; applications can now rely on this or select 1Kbyte units with the –**k** option.

diff — compare two files

Issue 4: The –**h** (half-hearted) option is omitted. Since it is marked PI in Issue 3 and since a POSIX-conformant *diff* does not have arbitrary file-length limitations, it must be omitted from portable applications.

The –**b** option is incorrectly described in Issue 3. The Issue 4 description, which involves all varieties of white space, not just blank characters, correctly matches historical implementations.

For applications that parse the output of *diff* –**e** or –**f**, note that the *ed* substitute command can now appear in the output; not all systems used this in the past.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

dircmp — directory comparison (TO BE WITHDRAWN)

Issue 4: The capability of *dircmp* is stabilised at the Issue 3 level, with the exception of the possible effects of the internationalisation variables on some systems. The *diff* utility with the –**R** option is recommended as a replacement for *dircmp*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

Input lines in any files being compared with the –**d** option are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

dirname — return directory portion of pathname

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The behaviour of *basename* and *dirname* in the XCU specification are coordinated so that when *string* is a valid pathname:

```
$(basename "string")
```

is a valid filename for the file in the directory:

```
$(dirname "string")
```

dis — disassembler (DEVELOPMENT) (OPTIONAL) (TO BE WITHDRAWN)

Issue 4: The *dis* utility will be withdrawn from a future issue. It has never been used portably.

Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**F** *function* and –**l** *string*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

du — estimate file space usage

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The –**a** and –**s** options are mutually exclusive. Some systems may produce output for –**sa**, but a fully portable application cannot use that combination.

The new –**x** option can be used to search a single file system for disk usage.

The previous behaviour of not listing non-directories explicitly given as operands, unless the –**a** option is specified, is reversed.

In the past, some systems did not produce error messages about inaccessible directories and files, so the –**r** option was used to force that. This behaviour is now the default and –**r** will be retired in the future. To suppress error messages, use shell redirection of standard error.

Previous issues did not require that space figures be reported uniformly in 512-byte units; applications can now rely on this or select 1Kbyte units with the –**k** option.

echo — write arguments to standard output

Issue 4: Any usage of *echo* where it does not control the format of its arguments should be replaced by the new *printf* utility. For example, usage such as:

```
echo Usage: ...
echo cannot read input file
```

remains portable, but the following examples are not:

```
echo Argument $1 not valid.
echo cannot read input file $2
```

The sequence \0 is replaced by a binary zero; previous issues incorrectly implied that it would be the two characters \ and 0.

ed — edit text

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. (This is considerably larger than some historical *ed* implementations have allowed.) Input files must end with a newline character.

Applications using *ed* − should be converted to use *ed* −**s**.

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 XCU specification 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 is to precede it with another character and then use the substitute command to delete that character.

The manner in which the **l** command writes non-printable characters is 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 character ‘‘a’’, a byte with a binary value of 1, and a character 1. In the format required in the XCU specification, a backslash appearing in the line is written as \\ so that the output is truly unambiguous. The method of marking the ends of lines is adopted from the *ex* editor and is required for any line ending in space characters; the $ is placed on all lines so that a real $ character at the end of a line cannot be misinterpreted.

Some historical implementations returned exit status zero even if command errors had occurred; this is not allowed by the XCU specification.

egrep — search a file with an ERE pattern

Issue 4: See *grep*.

env — set environment for command invocation

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *name* or *utility* operand beginning with −).

Applications using *env* − should be converted to use *env* −**i**.

ex — text editor

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**t** *tagstring*, −**c** *command* and −**w** *size*. The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input file lines must end with a newline character except for the final line.

The paragraph in INPUT FILES, ‘‘By default, ...,’’ is intended to close a long-standing security hole in *ex* and *vi*, that of the little-known *modelines*. This is the feature whereby a line in the first or last five lines of the file containing the strings "ex:" or "vi:" (and, apparently "ei:" or "vx:") is considered to be a line containing editor commands, and *ex* interprets all the text up to the next colon (:) or newline character as a command. Consider the consequences, for example, of an unsuspecting user using *ex* or *vi* as their editor when replying to a mail message in which a line such as:

```
"ex:! rm −rf *:"
```

appeared in the signature lines. Implementation extensions are sometimes available to allow this behaviour.

The **ve** version command, marked as MV in Issue 3, is omitted.

The default for editor option **exrc** is reversed from common practice to close a security hole. Those people who use the **.exrc** file in directories other than *$HOME* can retain the old behaviour, by setting this option in their *EXINIT* environment variables or their **$HOME/.exrc** files. Note that **noexrc** disables the use of **./.exrc**; **exrc** enables it. The default is to disable it.

The non-printable characters printed by the **list** command can no longer be relied upon to be in the form *letter* because this does not meet the requirement that the output line be unambiguous.

The following options are now considered to be extensions: *lisp* mode, the use of terminal function keys with **map**, and the **directory**, **edcompatible**, **redraw** and **slowopen** editor options.

expand — convert tabs to spaces

Issue 4: A new utility for Issue 4.

expr — evaluate arguments as an expression

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, an operand beginning with −).

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.

false — return false value

Issue 4: This utility operates exactly as in Issue 3.

fc — process command history list

Issue 4: A new utility for Issue 4.

fg — run jobs in the foreground

Issue 4: A new utility for Issue 4.

fgrep — search a file for a fixed-string pattern

Issue 4: See *grep*.

file — determine file type

Issue 4: A new utility for Issue 4.

find — find files

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

An application using *find* and expecting no output because none of −**print**, −**exec** or −**ok** were given will have to redirect standard output to **/dev/null**.

Applications using octal notation with −**perm** should be converted to use symbolic modes for maximum portability.

fold — filter for folding lines

Issue 4: A new utility for Issue 4.

fort77 — FORTRAN compiler (FORTRAN)

Issue 4: A new utility for Issue 4.

gencat — generate a formatted message catalogue

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *catfile* name beginning with −).

Input lines in the message text source file are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

get — get a version of an SCCS file (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**c** *cutoff*, −**l** *list* and −**x** *list*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

Applications using *get* −**lp** should be converted to use *get* −**L**.

getconf — get configuration values

Issue 4: A new utility for Issue 4.

Issue 4, Version 2:

The following changes have been made to the *system_var* table:

- Names beginning with POSIX_ have been changed to begin with _POSIX_.
- Names beginning with XOPEN_ have been changed to begin with _XOPEN_.
- MN_NMAX is changed to NL_NMAX.[1]
- NL_SET_MAX is changed to NL_SETMAX.
- NL_TEXT_MAX is changed to NL_TEXTMAX.
- The _XOPEN_CRYPT, _XOPEN_ENH_I18N and _XOPEN_SHM configuration variables have been added to the list.
- The ATEXIT_MAX, IOV_MAX, PAGESIZE, PAGE_SIZE and _XOPEN_UNIX configuration variables have been added to the list.

1. References in the XCU specification description of *getconf* linking MN_NMAX to NL_MAX are erroneous and will be corrected in a corrigendum.

getopts — parse utility options

Issue 4: A new utility for Issue 4.

The *getopts* command parses command line options appropriate to the new Utility Syntax Guidelines (refer to Section 2.9.1 on page 31). Shell scripts developed on traditional UNIX systems that used *getopt* will need to convert to using *getopts*.

grep — search a file for a pattern

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

All the utilities in the *grep* family are consolidated and take consistent selections of arguments across *grep*, *egrep* and *fgrep*. This results in adding −**f** to *grep*, −**e** to *grep* and *fgrep* (and allowing multiple patterns in the option-argument, separated by newline characters), and −**x** to *grep* and *egrep*. The new −**q** is added to all.

Historical implementations usually silently ignored all but one of the multiply-specified −**e** and −**f** options, but were not consistent as to which specification was actually used.

A definition of action taken when given a null RE or ERE is specified. This is an error condition in some historical implementations.

hash — remember or report utility locations

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a utility name beginning with −).

Some implementations of the shell do not remember or report utilities found through normal command search.

head — copy the first part of files

Issue 4: A new utility for Issue 4.

iconv — codeset conversion

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument or for: −**f** *fromcode* and −**t** *tocode*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

id — return user identity

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *user* name beginning with –).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

jobs — display status of jobs in the current session

Issue 4: A new utility for Issue 4.

join — relational database operator

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument or for: −**a** *file_number*, −**e** *string*, −**j1** *field*, −**j2** *field*, −**o** *list*, −**t** *char*, −**v** *file_number*, −**1** *field* and −**2** *field*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

A new field is allowed in the −**o** list to represent the join field, allowing the *full join* or *outer join* operations described in relational database literature.

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Applications using *join* with one of the variants of −**j** should be converted to use *join* −**1** and −**2**. Multi-argument *list* values for −**o** should be converted to a single argument. For example, convert the first of these commands to the second:

```
join -j1 3 -j2 4 -o 1.2 3.4 4.5 file1 file2
join  -1 3  -2 4 -o 1.2,3.4,4.5 file1 file2
```

kill — terminate or signal processes

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**s** *signal*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *pid* beginning with –).

Applications should migrate to the symbolic signal names with the −**s** option. For example, convert either of the first two commands to the third:

```
kill -9 1234
kill -kill 1234
kill -s kill 1234
```

lex — generate programs for lexical tasks (DEVELOPMENT)

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

None of the statistics or error messages have specified output formats; applications relying on one implementation's formats are not portable.

Applications cannot rely on receiving the summary of statistics by including a table size (for example, %p, %n) in the program. They must use **–v** to receive the summary. In any case, the summary and error messages may appear on either standard output or standard error, unless the **–t** option is used (in which case standard error is used).

New features are added that do not appear on all historical systems:

- The %x specifier for exclusive start conditions.
- The %array and %pointer declarations. X/Open systems have historically used an array for *yytext*, but a portable application cannot rely on this choice unless it codes %array. In particular, multi-file programs with external references to *yytext* outside the scanner source file require one of the new declarations to be considered strictly portable.
- The definition of the **input** function is changed to allow buffering of input.

line — read one line (TO BE WITHDRAWN)

Issue 4: The capability of *line* is stabilised at the Issue 3 level. The *read* utility is replacing *line*.

lint — check C-language programs (DEVELOPMENT) (TO BE WITHDRAWN)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: **–D** *name***[**=*value***]**, **–I** *directory*, **–l** *x*, **–L** *directory*, **–o** *x* and **–U** *name*. The **––** argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

ln — link files

Issue 4: The **––** argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The version of *ln* in Issue 3 unlinks the destination file, if it exists, by default. If the mode does not permit writing, that version prompts for confirmation before attempting the unlink. In this Issue 3 version, the **–f** option causes *ln* not to 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 has to be). The POSIX-2 standard does not allow the *ln* utility to unlink existing destination paths by default. Applications requiring the forced linking behaviour must use the **–f** option; those requiring the prompting features of *ln* need to be rewritten using *test*, *rm* and shell script prompting, such as with *read*.

locale — get locale-specific information

Issue 4: A new utility for Issue 4.

localedef — define locale environment

Issue 4: A new utility for Issue 4.

logger — log messages

Issue 4: A new utility for Issue 4.

logname — return user's login name

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

lp — send files to a printer

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**d** *dest*, –**o** *option* and –**t** *title*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The interaction between *PRINTER* and *LPDEST* is described.

lpstat — report line printer status information

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

ls — list directory contents

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Issue 3 indicates that not all systems use 512-bytes as reporting units. Because of POSIX compliance, all systems now use 512-byte units for reporting directory sizes. But, as explained in **APPLICATION USAGE**, portable applications should always use the exact byte counts, not block counts, to determine file sizes.

m4 — macro processor (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**D** *name***[**=*val***]** and –**U** *name*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

mail — send or read mail (TO BE WITHDRAWN)

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *name* beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The capability of *mail* is stabilised at the Issue 3 level. The *mailx* utility is replacing *mail*.

mailx — process messages

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**s** *subject* and −**u** *user*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, an address beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. The Subject field is now documented to be limited to {LINE_MAX} − 10 bytes and cannot contain a newline character. Input files must end with a newline character.

The −**h** *number* (number of network ‘‘hops’’ made so far) and −**r** *address* (pass *address* to network delivery software) options are omitted from Issue 4. Both are marked as PI and UN in Issue 3 and are not generally useful to portable applications. The **conv=***conversion* variable is omitted because it is not fully specified in Issue 3.

CW The *mailx* variables beginning with lower-case letters are no longer affected by shell or environment variable contents. Any settings of these variables must be accomplished by the *mailx* **set** command in one of the startup files.

The **allnet**, **onehop**, **sendmail** and **sendwait** variables are now considered to be extensions.

The **Copy**, **echo**, **Save**, **followup** and **Followup** commands and the −**F** option are now considered to be extensions.

The **version** command is marked MV in Issue 3 and is removed from Issue 4.

The default for **print**, **Print**, **type** and **Type** is now *more*, rather than the previous *pg*. The **Print** and **Type** commands are now affected by the pagination as well as their lower-case counterparts.

make — maintain, update and regenerate groups of programs (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument or for: −**f** *makefile*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a target name beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Portable makefiles require the **.POSIX** special target.

CW Commands that begin with a plus-sign (+) are executed even if the −**n** option is present. The behaviour of −**n** when the plus-sign prefix is encountered is extended

to apply to –**q** and –**t** as well. However, the System V and Issue 3 convention of forcing command execution with –**n** when a target's command line contains either of the strings $(MAKE) or ${MAKE} is not included in the portable makefile behaviour. The danger of this approach is illustrated with the following example of a portion of a makefile:

```
subdir:
    cd subdir; rm all_the_files; $(MAKE)
```

The command-line plus-sign prefix can provide the desired capability.

The –**S** option is added as an opposite of –**k**.

The **.PRECIOUS** special target is extended to affect all targets globally (by specifying no prerequisites). The **.IGNORE** and **.SILENT** special targets are extended to allow prerequisites.

Issue 4, Version 2:

Under the **Default Rules**, the string `-G$@` is deleted from the line referencing `sccs`.

man — display system documentation

Issue 4: A new utility for Issue 4.

mesg — permit or deny messages

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

mkdir — make directories

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**m** *mode*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

mkfifo — make FIFO special files

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**m** *mode*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

more — display files on a page-by-page basis

Issue 4: A new utility for Issue 4.

mv — move files

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The *mv* utility can now move directories across file system boundaries.

The −**i** option is added for interactive prompting prior to overwriting existing files.

Historic implementations of *mv* do not exit with a non-zero exit status if they are unable to duplicate any file characteristics when moving a file across file systems, nor do they write a diagnostic message for the user. The former behaviour is preserved to prevent scripts from breaking; a diagnostic message is now required, however, so that users are alerted that the file characteristics have changed.

newgrp — change to a new group

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a group name beginning with −).

nice — invoke a utility with an altered system scheduling priority

Issue 4: A new utility for Issue 4.

nl — line numbering filter

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument or for: −**b** *type*, −**d** *delim*, −**f** *type*, −**h** *type*, −**i** *incr*, −**l** *num*, −**n** *format*, −**s** *sep*, −**v** *startnum* and −**w** *width*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Applications that rely on placing options after the *file* operand should be converted to use the order prescribed by the Utility Syntax Guidelines.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

nm — write the name list of an object file

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The default output base for numeric quantities remains decimal, but this is now an extension to POSIX. A fully portable application should use −**t d** to achieve the decimal base.

The −**V** (output utility version) is marked as non-portable in Issue 3 and is omitted from Issue 4.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

nohup — invoke a utility immune to hangups

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a utility name beginning with −).

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.

od — dump files in various formats

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**A** *address_base*, −**j** *skip*, −**N** *count* and −**t** *type_string*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Portable applications should migrate from the −**bcdosx** options to −**t** *type_string* and the *offset* operand to −**A** *address_base*.

Issue 4, Version 2:
The description of the −**c** option is made dependent on the current setting of the LC_CTYPE category, and a reference to the POSIX locale is deleted.

pack — compress files (TO BE WITHDRAWN)

Issue 4: Conformance to the Utility Syntax Guidelines requires **[−]** to be treated as an operand, rather than an option. Thus, the previously accepted:

```
pack - -f file
```

should be changed to:

```
pack -f - file
```

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

In Issue 3, the exit status reflects the number of files that could not be packed. The new exit status description does not mandate that and applications relying on it have to be redesigned; they are probably not portable on Issue 3 systems anyway, because the exit status would wrap back to zero when 256 files could not be packed.

The *compress* utility is replacing *pack*. The two utilities are generally equivalent when used with *file* operands, but any part of the application that relies on a **.z** suffix has to be changed to handle **.Z**.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

Issue 4, Version 2:
The **DESCRIPTION** section is clarified to state that the ownership, modes, access time and modification time of the original file are preserved if the invoking process has appropriate privileges.

paste — merge corresponding or subsequent lines of files

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**d** *list*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines cannot contain NUL characters. Input files must end with a newline character.

patch — apply changes to files

Issue 4: A new utility for Issue 4.

pathchk — check pathnames

Issue 4: A new utility for Issue 4.

pax — portable archive interchange

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**b** *blocksize*, –**f** *archive*, –**o** *options*, –**p** *string*, –**s** *replstr* and –**x** *format*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The following pairs of commands demonstrate conversions from *cpio* and *tar* to *pax*. In all cases, the examples show comparable command-line usage rather than identical output formats. The –**x** option can be added to the *pax* commands shown, producing archives to select specific output formats:

```
ls * | cpio -ocv
pax -wdv *

find /mydir -type f -print | cpio -oc
find /mydir -type f -print | pax -w

cpio -icdum < archive
pax -r < archive

(cd /fromdir;find . -print) | cpio -pdlum /todir
pax -rwl /fromdir /todir

tar cf archive *
pax -w -f archive *

tar xfv - < archive
pax -rv < archive

(cd /fromdir; tar cf – . ) | (cd /todir; tar xf –)
pax -rw /fromdir /todir
```

The *cpio* archive format created when –**c** is not used is not necessarily supported by *pax*; *pax* may be able to read it, but cannot write it unless triggered by an implementation extension.

pcat — expand and concatenate files (TO BE WITHDRAWN)

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

In Issue 3, the exit status reflected the number of files that could not be unpacked. The new exit status description does not mandate that and applications relying on it will have to be redesigned; they were probably not portable before anyway, because the exit status would have wrapped back to zero when 256 files could not be packed.

The *zcat* utility is replacing *pcat*. The two utilities are generally equivalent when used with *file* operands, but any part of the application that relies on a **.z** suffix has to be changed to handle **.Z**.

Issue 4, Version 2:
The **DESCRIPTION** section no longer specifies the assertion that a file is not written if the filename has more than {NAME_MAX}−2 bytes.

pg — file perusal filter for soft-copy terminals (TO BE WITHDRAWN)

Issue 4: Partial conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**p** *string*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The capability of *pg* is stabilised at the Issue 3 level. The *more* utility is replacing *pg*.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

pr — print files

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument or for: −**h** *header*, −**l** *lines*, −**o** *offset* and −**w** *width*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Historical systems accepted the *header* string as the next argument, such that the following were equivalent:

```
pr -3dh "file list" file1 file2
pr -h3d "file list" file1 file2
```

but the Utility Syntax Guidelines prevent this usage. Use the version shown in **EXAMPLES**.

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 characters on each page to a form-feed character and, if standard output is a TTY device, sends an alert character to standard error and

reads a line from **/dev/tty** before the first page. Therefore, the use of –**f** is marked as an extension and the –**F** option is added.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

printf — write formatted output

Issue 4: A new utility for Issue 4.

prs — print an SCCS file (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**c** *cutoff*, –**d** *dataspec* and **[**–**r [***SID***]]**. In Issue 3, the –**c** and –**d** options are listed with optional option-arguments; applications must provide these option-arguments.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

ps — report process status

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**g** *grouplist*, –**G** *grouplist*, –**n** *namelist*, –**o** *format*, –**p** *proclist*, –**t** *termlist*, –**u** *userlist*, –**U** *userlist*.

The new –**o** option can be used to customise the output. For applications requiring only certain pieces of information from *ps*, this may be preferable to parsing the –**f** or –**l** formats. Fully portable applications should convert to the use of –**o**.

pwd — return working directory name

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

read — read a line from standard input

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a variable name beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

red — restricted text editor (WITHDRAWN)

Issue 4: This utility is withdrawn because it does not provide the secure environment it implies.

renice — set system scheduling priorities of running processes

Issue 4: A new utility for Issue 4.

rm — remove directory entries

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The −**R** option is added for symmetry with other utilities, but requires no modifications to existing applications.

Some systems were inconsistent about prompting to standard output or standard error; the XCU specification requires consistent use of standard error.

rmdel — remove a delta from an SCCS file (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**r** *SID*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

rmdir — remove directories

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

On some previous 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 **STDERR** section 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.

sact — print current SCCS file-editing activity (DEVELOPMENT)

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

Issue 4, Version 2:
The **STDERR** section encompasses informative messages concerning sccs files with no impending deltas.

sccs — front end for the SCCS subsystem (DEVELOPMENT)

Issue 4: A new utility for Issue 4.

sdb — symbolic debugger (WITHDRAWN)

Issue 4: This utility has never been used in a portable manner. Software developers should use the debuggers available on each system, many of which are more powerful and easier to use than *sdb*.

sed — stream editor

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**e** *script* and –**f** *script_file*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

This document requires implementations to support at least ten distinct *wfiles*, matching historical practice on many implementations. This is an extension to the POSIX-2 standard limit of nine. Applications should limit themselves to nine *wfiles* for portability outside XSI-conformant systems.

The manner in which the *l* command writes non-printable characters is changed to avoid the historical backspace-overstrike method and other requirements are added to achieve unambiguous output.

The requirements for acceptance of blank characters and space characters in command lines is made more explicit to describe existing practice clearly and remove confusion about the phrase ''protect initial spaces and tabs from the stripping that is done on every script line'' which appeared in Issue 3. (Not all implementations are known to strip blank characters from text lines, although they all allow leading blank characters preceding the address on a command line.)

The treatment of **#** comments differs from Issue 3, which only allows a comment as the first line of the script. The comment character is treated as a command and it has the same properties in terms of being accepted with leading blank characters.

Issue 3 also requires that a *script_file* have at least one non-comment line. Some historical implementations behave in unexpected ways if this is not the case. A correct Issue 4 implementation permits a *script_file* that consists only of comment lines.

CW The treatment of the **p** flag to the **s** command differs between historical systems 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'
```

the POSIX-2 standard, 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 and Issue 3 systems write the **A** only once in both examples, because the **p** flag is ignored if the –**n** option is not specified.

The form of the substitute command that uses the *n* suffix is limited by Issue 3 to the first 512 matches. This limit is removed in Issue 4 because there is no reason why 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 the character a on a line.

sh — shell, the standard command language interpreter

Issue 4: The – or –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

See also Chapter 3 on page 37.

sleep — suspend execution for an interval

Issue 4: The exit status is allowed to be zero when *sleep* is interrupted by the SIGALRM signal.

sort — sort, merge or sequence check text files

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**k** *keydef*, –**o** *output*, –**t** *char*, –**z** *recsz*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files should end with a newline character.

Applications should migrate to the use of the –**k** option; see **EXAMPLES** for conversion examples.

The –**y**[*kmem*] option in Issue 3 is omitted and the –**z** *recsz* option is marked as not supported on all systems. Portable applications should specify neither. The –**z** option is not standard practice on most systems, and is inconsistent with using *sort* to sort several files individually and then to merge them together. The description of preventing abnormal termination is not applicable to any X/Open system and –**z** is maintained only for temporary backwards compatibility. The –**y** option is removed because it could not be used portably.

Examples in some historical documentation state that options –**um** with one input file keep the first in each set of lines with equal keys. This behaviour is deemed to be an implementation artifact and is not made standard.

Issue 3 indicates that ‘‘setting –**n** implies –**b**.’’ The description of –**n** already states that optional leading blank characters 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 (for example, to sort mod 100), that offset is measured relative to the most significant digit, not to the column. Therefore, the –**b** implication is omitted from the XCU specification and an application writer wishing to achieve the previously mentioned side effects has to code the –**b** option manually.

spell — find spelling errors (TO BE WITHDRAWN)

Issue 4: This utility is being withdrawn because there is no known technology that can be used to make it recognise general language for user-specified input without providing a complete dictionary and grammar along with the input file.

split — split files into pieces

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**a** *suffix_length* and –**l** *line_count*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

In this issue, *split* can now deal with binary files, can split at arbitrary byte boundaries and can create a larger number of output files.

strings — find printable strings in files

Issue 4: A new utility for Issue 4.

strip — remove unnecessary information from executable files (DEVELOPMENT)

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

All the options marked UN in Issue 3 are removed from Issue 4.

stty — set the options for a terminal

Issue 4: The SWTCH control character, marked as OP in Issue 3, is omitted.

The **loblk** control mode is omitted because *shl* is not supported on all systems.

The setting of a line discipline number via the **line** option is omitted because there is no underlying support for it in the POSIX-1 standard.

The Issue 3 description of Control Mode 0 says: ''Hang up line immediately. This applies to all terminal lines, not just modem lines. A SIGHUP signal is sent to all processes attached to the line.''. This description is incorrect and the Issue 4 description accurately reflects the processing of modem disconnects on historical systems.

sum — print checksum and block count of a file (TO BE WITHDRAWN)

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The capability of *sum* is stabilised at the Issue 3 level. The *cksum* utility is replacing *sum*.

tabs — set terminal tabs

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**T** *type*.

In Issue 3 it is stated that tab and margin setting are always performed via characters output to standard output. In Issue 4, acknowledgement is given that this is not necessarily the case for all terminal types.

tail — copy the last part of a file

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**c** *number* and –**n** *number*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Issue 3 described the interaction of the –**f** option and pipes, but not FIFOs. The –**f** option is not ignored when the standard input or a *file* operand is a FIFO.

Applications using one of the obsolescent forms should be converted to use –**c**, –**f** and –**n**. For example:

Convert from	Convert to
`tail -10`	`tail -n 10`
`tail -5c`	`tail -c 5`
`tail -3b`	`tail -c 1536`
`tail +10f`	`tail -f -n +10`

talk — talk to another user

Issue 4: A new utility for Issue 4.

tar — file archiver (TO BE WITHDRAWN)

Issue 4: The capability of *tar* is stabilised at the Issue 3 level. The *pax* utility is replacing *tar* and is able to read and write archives produced by *tar*.

tee — duplicate standard input

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Some historical implementations ignore write errors. This is explicitly not permitted by the XCU specification.

Some historical implementations use O_APPEND when providing append mode; others use the *lseek*( ) function to search to the end of the file after opening the file without O_APPEND. This document requires capability equivalent to using O_APPEND.

test — evaluate expression

Issue 4: See **APPLICATION USAGE** for details of converting the –**o** and –**a** binary primaries for maximal portability. The following list of paired commands shows sample conversions:

```
test $1 = cat -o $2 = mouse -a $3 = bird
test "$1" = "cat" || { test "$2" = "mouse" && test "$3"
    = "bird"; }

test $1 = red -a $2 = white -o $3 = blue
test "$1" = "red" && test "$2" = "white" || test "$3"
    = "blue"
```

```
test $1 = car -a \( $2 = boat -o $3 = plane \)
test "$1" = "car" && ( test "$2" = "boat" || test "$3"
    = "plane" )
```

In the preceding conversions, the **{}** and **()** grouping forms accomplish the same result, but the latter creates a subshell. This may cause a slight performance penalty on some systems.

The new –**e** primary is 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.

The –**t** *file_descriptor* primary is shown with a mandatory argument because the grammar is ambiguous if it can be omitted. Issue 3 allows it to be omitted, providing a default of 1.

time — time a simple command

Issue 4: In Issue 3 it is unclear whether *time* operates on a simple command (utility name plus arguments) or on a complex shell command, such as a pipeline. In Issue 4, it is now clear that only a simple command can be timed portably, because the pipeline cases are now explicitly unspecified.

The following example command can be used to apply *time* portably to a complex command:

```
time sh -c 'complex-command-line'
```

touch — change file access and modification times

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**r** *ref_file* and –**t** *time*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

An ambiguity exists in the Issue 3 specification when a pathname that is a decimal number leads the operands; it is treated as a time value. A portable application must use the –**t** *time* construct to avoid this ambiguity if it cannot control the filenames in use.

At least one historical implementation of touch incremented the exit code if –**c** is specified and the file does not exist. This document requires exit status zero if no errors occur.

tput — change terminal characteristics

Issue 4: A new utility for Issue 4.

tr — translate characters

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *string1* value beginning with −).

According to Issue 3, a range expression required enclosing square-brackets, such as:

```
tr '[a-z]' '[A-Z]'
```

However, BSD-based systems do not require the brackets and this convention is used by the XCU specification to avoid breaking large numbers of BSD scripts:

```
tr a-z A-Z
```

The preceding Issue 3 script continues to work because the brackets, treated as regular characters, are translated to themselves. However, any Issue 3 script that relies on:

```
a-z
```

representing the three characters a, −, and z has to be rewritten as:

```
az-
```

Better yet, most range expressions should be converted to use the character classification methods, such as [:alpha:] or [:upper:].

NUL characters are no longer stripped from the input automatically. See **APPLICATION USAGE**.

The use of octal values to specify control characters is not portable. The introduction of escape sequences for control characters provides the necessary portability, but applications should be checked for usage such as:

```
tr '\b' B
```

which formerly meant to translate b to B, but now translates the backspace character to B. The previous ability to escape any character with a backslash is removed and unspecified results occur.

Issue 3 support for multi-character collating elements, using the:

```
[[.cc.]]
```

construct, and implicitly in a range expression, is removed.

In Issue 3, the:

```
[:class:]
```

and:

```
[=equiv=]
```

conventions are shown with double brackets, as in regular expression syntax. Because *tr* does not implement regular expression principles, but just borrows part of the syntax, the double brackets are removed from the POSIX-2 standard *tr*. Applications must remove the extra brackets.

Examples 1 and 5 in the Issue 3 *tr* are no longer valid. Also, note the differences in Issue 3 example number 3 and its replacement, Issue 4 number 2.

true — return true value

Issue 4: This utility operates exactly as in Issue 3.

tsort — topological sort

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

tty — return user's terminal name

Issue 4: Applications using *tty* −**s** should be converted to use *test* −**t 0**.

type — write a description of command type

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a command name beginning with −).

A fully portable application should use *command* −**V** instead of *type*.

ulimit — set or report file size limit

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

umask — get or set the file mode creation mask

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a *mode* operand beginning with −, such as −**r**, −**w** or −**x**).

Applications should migrate to the symbolic permissions form.

The default output format is unspecified. Portable applications should use −**S**.

unalias — remove alias definitions

Issue 4: A new utility for Issue 4.

uname — return system name

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

Issue 4, Version 2:
The **SYNOPSIS** section now lists all valid options.

uncompress — expand compressed data

Issue 4: A new utility for Issue 4.

Issue 4, Version 2:
The **DESCRIPTION** section is clarified to state that the ownership, modes, access time and modification time of the original file are preserved if the invoking process has appropriate privileges.

unexpand — convert spaces to tabs

Issue 4: A new utility for Issue 4.

unget — undo a previous get of an SCCS file (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**r** *SID*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

uniq — report or filter out repeated lines in a file

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**f** *fields* and –**s** *chars*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Applications using *uniq* –*n* or +*n* should be converted to use –**f** *fields* and –**s** *chars*, for example:

Convert from	Convert to
`uniq -3`	`uniq -f 3`
`uniq +3`	`uniq -s 3`

unpack — expand files (TO BE WITHDRAWN)

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

In Issue 3, the exit status reflected the number of files that could not be unpacked. The new exit status description does not mandate that and applications relying on it will have to be redesigned; they were probably not portable before anyway, because the exit status would have wrapped back to zero when 256 files could not be packed.

The *uncompress* utility is replacing *unpack*. The two utilities are generally equivalent when used with *file* operands, but any part of the application that relies on a **.z** suffix has to be changed to handle **.Z**.

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

Issue 4, Version 2:

The **DESCRIPTION** section is clarified to state that the ownership, modes, access time and modification time of the original file are preserved if the invoking process has appropriate privileges, and the **DESCRIPTION** section no longer specifies the assertion that a file is not written if the filename has more than {NAME_MAX}–2 bytes.

uucp — system-to-system copy

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**n** *user*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

In Issue 3 the *uucp* utility always assigns universal read and write permissions to files (666) and preserves execute permissions across the transmission. In Issue 4 the permissions are implementation-dependent. Therefore, it is no longer possible to assume that a portable application will be able to access files on the other system after transfer.

uudecode — decode a binary file

Issue 4: A new utility for Issue 4.

uuencode — encode a binary file

Issue 4: A new utility for Issue 4.

uulog — query system-to-system transaction log

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**s** *system*.

uuname — list names of other known uucp systems

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables.

uupick — receive public system-to-system file copies

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**s** *system*.

uustat — uucp status inquiry and job control

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**k** *jobid*, –**r** *jobid*, –**s** *system* and –**u** *user*.

uuto — send public system-to-system file copies

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

uux — remote command execution

Issue 4: Applications using *uux* – should be converted to use *uux* –**p**.

val — validate SCCS files (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**m** *name*, –**r** *SID* and –**y** *type*. The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with –).

Some of the utility output formats are specified, whereas they are not specified in Issue 3. Applications relying on previous output formats are not portable.

vi — screen-oriented (visual) display editor

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**t** *tagstring*, –**c** *command* and –**w** *size*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input file lines must end with a newline character, except for the final line.

The **e** and **E** commands described here differ slightly from some historical implementations in that an empty or blank line is described as containing exactly one word or bigword, just as for **b**, **B**, **w** and **W**. On some systems, the **E** command skips over empty/blank lines. There is now a consistent definition of bigword between the **E** command and the others so that they all treat empty or blank lines the same way.

wait — await process completion

Issue 4: Multiple *pid* operands and job control job IDs are now allowed.

wall — write to all users (WITHDRAWN)

Issue 4: This utility is withdrawn because it cannot be used portably by an application. It is usually usable only by applications with appropriate privileges. Applications needing some sort of general broadcasting facility can use *who* and *write*, but the user community accessible in this way is limited by security considerations and the *mesg* status of each user at the time.

wc — word, line and byte count

Issue 4: The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with –).

The new –**m** option allows counting by characters (which can be multi-byte) instead of bytes.

Some historical implementations use only the space character, tab character and newline character as word separators, which would result in different counts for some files.

what — identify SCCS files (DEVELOPMENT)

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a directory name beginning with −).

who — display who is on the system

Issue 4: The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a filename beginning with −).

The forms *who am i* and *who am I* are being replaced by *who* −**m** for portable applications.

write — write to another user

Issue 4: This utility operates exactly as in Issue 3, except for the effects of the internationalisation variables. However, it is now acknowledged that the implementation may define special control characters as part of extensions to the terminal driver and that these will also be effective in the *write* session. For example, some systems have implemented <control>-W to back up one word in the input, similar to its use in *vi*.

xargs — construct argument list(s) and invoke utility

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: −**I** *replstr*, −**L** *number*, −**e** *eofstr*, −**n** *number* and −**s** *size*.

The −− argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a utility name beginning with −).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

The −**e** option is changed to support the Utility Syntax Guidelines and because the POSIX-2 standard does not allow the default use of underscore as a logical EOF character. In Issue 3, underscore is the default and −**e** without an option-argument turns off all logical EOFs. Portable applications requiring the former default case must specify −**e** **_**.

The −**i** and −**l** options are declared obsolescent because they do not conform to the Utility Syntax Guidelines. They should be replaced as follows:

Convert from	Convert to
`-i`	`-r` or `-I {}`
`-i`*`replstr`*	`-I` *`replstr`*
`-l`	`-L 1`
`-l`*`number`*	`-L` *`number`*

yacc — yet another compiler compiler (DEVELOPMENT)

Issue 4: Conformance to the Utility Syntax Guidelines requires blank-character separation of the option and argument for: –**b** *file_prefix* and –**p** *sym_prefix*.

The –– argument is required to delimit the first operand if that operand could be misinterpreted as an option (for example, a grammar name beginning with –).

Input lines are now documented to be limited to {LINE_MAX} bytes and cannot contain NUL characters. Input files must end with a newline character.

Simultaneous execution of multiple invocations of *yacc* and multiple parsers in the same file are now possible, given the new –**b** and –**p** options.

zcat — expand and concatenate data

Issue 4: A new utility for Issue 4.

XPG3-XPG4 Base Migration Guide, Version 2

Part 3:

System Interfaces and Headers Migration

X/Open Company Ltd.

Chapter 5

Program Migration and Portability

This chapter covers a general set of topics of interest to developers:

- migrating applications from implementations conforming to Issue 3 to implementations conforming to Issue 4, Version 2
- migrating applications originally developed on traditional UNIX systems to systems that conform to XPG4 UNIX.

Specific topics covered include:

- feature groups
- the compilation environment
- functional duplication in the Single UNIX Specification
- other programming considerations
- interprocess communications
- STREAMS
- error codes
- makefile portability.

5.1 Feature Groups

The concept of feature groups was introduced in Issue 4. The majority of functions belong to the BASE capability. These functions are mandatory on XSI-conforming systems; applications can rely on their existence.

In addition to the BASE capability, there are five feature groups as follows:

- Encryption; identified as CRYPT on reference pages
- Enhanced Internationalisation; identified as ENHANCED I18N on reference pages
- POSIX.2 C-language Binding; identified as POSIX2 CLB on reference pages
- Shared Memory; identified as SHARED MEM on reference pages
- X/Open UNIX Extension; identified as X/OPEN UNIX on reference pages.

The feature group identifier appears at the top of appropriate reference pages, and BASE is used to identify all functions belonging to the BASE capability.

The XPG4 XSH specification defines which feature groups are mandatory and which are optional for specific profiles or component definitions. Refer to Section 1.7 on page 11 for an overview of XPG4 profiles. Some interfaces in feature groups also define optional behaviour. Developers must refer to a product's Conformance Statement to determine what feature groups are supported on that particular implementation.

Feature test macros are defined by an implementation and available for the application source-code to check to determine whether a feature group is supported or not at compile time.

Feature Test Macro	Feature Group
_XOPEN_CRYPT	Encryption
_XOPEN_ENH_I18N	Enhanced Internationalisation
_POSIX2_C_VERSION	POSIX.2 C-language Binding
_XOPEN_SHM	Shared Memory
_XOPEN_UNIX	X/Open UNIX Extension

Table 5-1 Feature Test Macros and Feature Groups

With two exceptions, the implementation shall define the feature test macro to be other than –1 if the feature group is provided, and –1 if the feature group is not implemented. The exceptions are:

- The _POSIX2_C_VERSION macro needs to be defined explicitly in **<unistd.h>**.
- An implementation not supporting the X/Open UNIX Extension need not define *_XOPEN_UNIX* to be –1. (The system may pre-date the introduction of the X/Open UNIX feature group.)

5.2 The Compilation Environment

A section is added to Issue 4 indicating use of the feature test macro _XOPEN_SOURCE. In summary, ensure that this macro is defined before the inclusion of any header defined in the XSH specification; its inclusion subsumes the use of _POSIX_SOURCE and _POSIX_C_SOURCE. This macro causes all of the proper symbols to be exposed with respect to base functionality and any of the XSH specification feature groups that are provided on that particular XPG4 branded system.

The Single UNIX Specification introduces the X/Open UNIX Extension feature group, which requires the additional macro _XOPEN_SOURCE_EXTENDED to be defined to be 1 prior to the inclusion of the first header. These two macros, _XOPEN_SOURCE and _XOPEN_SOURCE_EXTENDED, are set by the application to ensure the identifiers and prototypes are correctly exposed, and should not be confused with the other feature test macros that are set by the implementation to be tested by the application. One (_XOPEN_SOURCE and _XOPEN_SOURCE_EXTENDED) is a header configuration mechanism used by the application to provide information to the implementation. The other (_XOPEN_UNIX) is an announcement mechanism provide by the implementation to be tested by the application source-code.

The _XOPEN_SOURCE macro may be defined automatically by the compilation environment, but to ensure maximum portability the application should define the macro either on the compilation command line, or best at the beginning of each source module prior to the inclusion of any headers. _XOPEN_SOURCE_EXTENDED will not be automatically defined by the compilation environment.

The X/Open Name Space has been fully defined.

Application developers are advised to study the XSH specification, Section 2.2, The Compilation Environment, when migrating or developing programs on XPG4 Base 95 or XPG4 UNIX branded systems.

5.3 Functional Duplication

Part of the development of the Single UNIX Specification involved merging three industry portability specifications and surveying existing applications. Some functionality is duplicated in some interfaces, because of this approach. This functional duplication was carefully managed in the specification, and serves the portability of existing applications originally developed for UNIX and UNIX-like environments. A number of things have been done with the format of the Single UNIX Specification to provide direction for new application development work, and to support the migration of applications that need to be portable to earlier versions of the Single UNIX Specification.

- Text shading and portability codes in the reference pages notify programmers of potential functionality that may reduce portability to other platforms.
- Interfaces that duplicate functionality and are not intended to be the preferred future direction:
  - — may be marked as **TO BE WITHDRAWN** on the **NAME** line to draw attention to areas that need attention
  - — have clearly marked migration paths in favour of current industry standards in the **APPLICATION USAGE** section of the reference page.

The following summarises the areas in the Single UNIX Specification where duplicated functionality exists. References to the recommended interfaces are found in the appropriate sections of Chapter 7 on page 121, and in the appropriate **APPLICATION USAGE** sections of the XSH specification.

- **<strings.h> versus <string.h>**

  There is direct overlap between the BSD byte-string functions (*bcmp*( ), *bcopy*( ), *bzero*( )) and the ISO C standard memory handling functions (*memcmp*( ), *memmove*( ), *memset*( )). Other string functions traditionally found on BSD systems are also covered by ISO C standard functions: *index*( ) maps to *strchr*( ) and *rindex*( ) maps to *strrchr*( ). The ISO C standard functions are the preferred method of performing these operations.

  The ISO C standard functions are all defined in **<string.h>**, while the BSD functions have been moved to **<strings.h>**.

- **Temporary File Creation**

  There are a number of interfaces to create temporary files and temporary filenames. The interfaces *tempnam*( ) and *tmpnam*( ) create filenames suitable for temporary files, and *tmpfile*( ) actually creates a temporary file for use.

  The functions *mkstemp*( ) and *mktemp*( ) are present in the Single UNIX Specification as part of the X/Open UNIX Extension, but only exist to support historical usage.

- **Regular Expression Handling**

  There are several ways to handle regular expressions and pattern matching:

  - — the newer more functional POSIX.2-defined functions *regcomp*( ) and *regexec*( ) included through **<regex.h>**
  - — the historical model covered by *compile*( ), *step*( ) and *advance*( ) and included through **<regexp.h>**
  - — the *regcmp*( ) and *regex*( ) functions as defined in **<libgen.h>**

— the *re_comp*( ) and *re_exec*( ) functions as defined in **<re_comp.h>**.

The POSIX.2 functionality is the preferred method of handling regular expressions, and the other methods are marked **TO BE WITHDRAWN**.

- **Multiple Signal Models**

  The development of the Single UNIX Specification brought together three separate signal models from POSIX.1, BSD and the SVID. While the BSD and SVID signal models are now supported on X/Open UNIX systems, supporting ports of historical applications, the newer POSIX.1-based model should be used to plan for future portability, and support portability to implementations that conform to an earlier version of the XSH specification. Reference pages point to equivalent calls in the POSIX.1 signal model in the **APPLICATION USAGE** sections, for the calls based on the other two models.

  If an application depends upon historical BSD signal behaviour, it will need to replace all calls to *signal*( ) with calls to *bsd_signal*( ), to avoid the overloading of the *signal*( ) call that occurred when the models came together in the specification.

- **Advisory Record Locking**

  Advisory record locking is supported with the *lockf*( ) interface, new to this version of the XSH specification, and also with the *fcntl*( ) interface. The latter method is preferred for applications that require portability to implementations conforming to earlier versions of the XSH specification.

- **mknod( ) and mkfifo( )**

  The *mknod*( ) interface has been added to the Single UNIX Specification but the only portable use of the function is for creating FIFO-special files. Because of this and the existence of *mkfifo*( ) as the POSIX.1 standard method of creating a FIFO-special file, developers are encouraged to use *mkfifo*( ) for future portability, and portability to implementations conforming to earlier versions of How to Brand — What to Buy.

- **Non-local Goto**

  The *_longjmp*( ) and *_setjmp*( ) interfaces (new in the Single UNIX Specification) are identical to the ISO C standard interfaces *longjmp*( ) and *setjmp*( ), with the exception that they do not manipulate the signal mask. Applications that depend upon the value of the signal mask should use the POSIX.1-specified *siglongjmp*( ) and *sigsetjmp*( ) interfaces.

- **Current Working Directory**

  The *getcwd*( ) interface should be used to determine the current working directory, rather than *getwd*( ), to plan for future portability, and support portability to implementations that conform to an earlier version of the XSH specification.

- **Text String to Number Conversion**

  Text strings representing numbers should be converted to numbers using the standard *strtod*( ) and *strtol*( ) interfaces, rather than the historical *atof*( ), *atoi*( ) and *atol*( ) functions. While these interfaces have been added to the Single UNIX Specification to support the porting of historical applications, the newer standards-based interfaces have better error checking.

  In a similar vein, the floating-point number to string conversion routines, *ecvt*( ), *fcvt*( ), *gcvt*( ) should be replaced with *sprintf*( ) for maximum portability.

- **utimes( ) versus utime( )**

  To set file access and modification times, a developer should use the *utime*( ) function, aligned with POSIX.1 and supported by implementations conforming to earlier versions of the XSH specification, rather than the *utimes*( ) interface.

5.4 Other Programming Considerations

Functional duplication should be of concern to programmers developing new applications, as they want to write the application to be as portable as possible over the long term. Programmers porting existing applications are less likely to be concerned about these issues immediately, but may discover a number of other issues effecting their portability.

5.4.1 Argument Type Changes

Argument type changes tend to be to more exact definitions, such as `int` to `mode_t` or `size_t`. Much of this is as a result of the C-language and POSIX standards efforts, but function prototypes not covered by either standard have been rationalised as well. For example, the second argument to *bzero*( ) changed from an `int` to a `size_t` from 4.3BSD to 4.4BSD, and the Single UNIX Specification reflects the newer more exact definition.

5.4.2 Prototype Changes and Movement

Certain function prototypes may have changed in subtle ways (types), or moved entirely to different header files. Prototypes moved from one header to another due once again to standards efforts. For example, the ISO C standard memory functions have historically appeared in **<memory.h>**, but moved to **<string.h>** with the standard. The BSD byte-string functions such as *bcopy*( ) and *bcmp*( ) have their prototypes defined in **<string.h>** in the 4.4BSD documentation, but **<strings.h>** in the Single UNIX Specification. Again, these moves are not strictly because of standards efforts. For example, the older regular expression handling functions *compile*( ), *step*( ), *advance*( ) moved from **<regexpr.h>** to **<regexp.h>** as part of the move to the Single UNIX Specification.

5.4.3 Process Environment Access

For C-language processes, the environment is an array of pointers to strings defined as:

```
extern char **environ;
```

C-language programs can use *getenv*( ) to determine the existence and value of environment variables. Environment variables can be placed in a program's environment using *putenv*( ), directly manipulating the *environ* array (not recommended), or using the *envp* arguments when creating a process (using *exec*( )). The *environ* array should not be directly accessed from the application.

System V platforms support a third argument to *main*( ) that is a null-terminated array of character string pointers to the environment. Applications that rely on this behaviour will need to be reworked to use the *environ* array, a two argument *main*( ), and possibly *getenv*( ) and *putenv*( ).

5.4.4 Pseudo-terminals

Issue 4, Version 2 introduces pseudo-terminals to the specification. This raises a portability problem for applications source-code that may have been developed on a BSD-based system and is being moved to a System V-based system, or *vice versa*. Pseudo-terminal implementations differ in these two operating environments with respect to initialisation.

The following example code (adapted from the module **SRC/userintf.c** in the VSX4 test suite) illustrates opening a pseudo-terminal to handle both environments.

Example 5-1 Pseudo-terminal Initialisation

```
/*
 * Openpty() opens the master and slave sides of a pseudo-terminal.
 * Both device names are supplied, but if master ptys are obtained
 * from a clone device the slave name is a dummy which must be
 * overwritten with the real device name.  If cntrl is true
 * the slave must be opened as a controlling terminal by calling
 * openctl() instead of plain open().  Both devices must be opened
 * for reading and writing.  On many systems master devices can
 * only be opened once, giving EBUSY for example on subsequent
 * opens.  If openpty() encounters a busy master device, it must
 * simply open the slave.
 *
 * The open file descriptors are returned via the pointers mfdp and
 * sfdp.  Openpty() returns 0 for success or -1 for failure.
 *
 */

#define CLONE_MPTY  /* delete this define if master pty's are not
                       obtained from a clone device (e.g. /dev/ptmx) */
#define STREAMS_PTY /* delete this define if pty's do not use STREAMS */

#ifdef STREAMS_PTY
#include <sys/stropts.h>
#endif

int
openpty(char *master, char *slave, int *mfdp, int *sfdp, int cntrl) {
    *mfdp = open(master, O_RDWR|O_NOCTTY);
    if (*mfdp < 0 && errno != EBUSY)
        return -1;

#ifdef CLONE_MPTY
    if (*mfdp >= 0)
    {
        char *newslave;
        size_t len;
        static struct { char *ptr; size_t len; } slavelen[2];
        extern char *ptsname();

        (void) grantpt(*mfdp);   /* change permission of slave */
        (void) unlockpt(*mfdp);  /* unlock slave */
        newslave=ptsname(*mfdp); /* get a slave */
        if (slavelen[0].ptr == NULL)
        {
            /* remember length of original dummy name
               to be used if called again */
            slavelen[0].ptr = slave;
            slavelen[0].len = strlen(slave);
        }
```

```
            else if (slave != slavelen[0].ptr && slavelen[1].ptr == NULL)
            {
                /* remember length of original dummy name
                   to be used if called again */
                slavelen[1].ptr = slave;
                slavelen[1].len = strlen(slave);
            }
            if (slave == slavelen[0].ptr)
                len = slavelen[0].len;
            else
                len = slavelen[1].len;
            if (strlen(newslave) > len)
            {
                (void) fprintf(stderr, "Error in openpty()\n");
                return -1;
            }
            (void) strcpy(slave, newslave);
        }
    #endif

        if (cntrl)
            *sfdp = openctl(slave, O_RDWR);
        else
            *sfdp = open(slave, O_RDWR|O_NOCTTY);

        if (*sfdp < 0)
            return -1;

    #ifdef STREAMS_PTY
        if (*mfdp >= 0)
        {
            (void) ioctl(*sfdp, I_PUSH, "ptem");   /* pseudo tty emulator */
            (void) ioctl(*sfdp, I_PUSH, "ldterm"); /* line discipline */
        }
    #endif

        return 0;
    }

    /*
     * Ptygetattr() obtains terminal attributes for the slave pseudo-
     * terminal corresponding to the master device addressed by mfd.
     * It returns 0 for success, -1 for failure.
     *
     */

    int
    ptygetattr(int mfd, struct termios *termios_p) {
    #ifdef STREAMS_PTY
        extern char *ptsname();
        char *slave;
        int ret, sfd = -1;
```

```
    /* Obtain slave device name and call tcgetattr() on the slave */
    slave = ptsname(mfd);
    if (slave && (sfd = open(slave, O_RDWR)) >= 0)
        ret = tcgetattr(sfd, termios_p);
    else
        ret = -1;
    if (sfd >= 0)
        (void) close(sfd);
    return ret;
#else
    /* This code assumes tcgetattr() on the master returns the
       settings of the slave (true for BSD-style pseudo-terminals) */
    return tcgetattr(mfd, termios_p);
#endif
}
```

5.5 Errors

5.5.1 Issue 4

The following symbolic name is added to the list of errors:

EILSEQ

This error is defined for use by certain of the WPI functions and indicates that a wide-character code does not correspond to a valid character in the prevailing codeset, or that a byte sequence does not form a valid wide-character code when converting from external to internal character mappings.

The only other change to the list of errors in Issue 4 is in the description of the [ENAMETOOLONG] error. Issue 3 indicates that this error is returned if a path component is longer than {NAME_MAX} and {_POSIX_NO_TRUNC} is in effect. For alignment with the FIPS requirements, {_POSIX_NO_TRUNC} must always be set on Issue 4 systems, hence the above caveat is removed.

5.5.2 Issue 4, Version 2

A substantial number of error codes appeared with the creation of the Single UNIX Specification. As well, some error values can be returned under new conditions. For example, a second [ENAMETOOLONG] condition is defined for many interfaces that may report excessive length of an intermediate result of pathname resolution of a symbolic link. New error values are listed below:

EADDRINUSE	EADDRNOTAVAIL	EAFNOSUPPORT	EALREADY
EBADMSG	ECONNABORTED	ECONNREFUSED	ECONNRESET
EDESTADDRREQ	EDQUOT	EHOSTUNREACH	EINPROGRESS
EISCONN	ELOOP	EMSGSIZE	EMULTIHOP
ENETDOWN	ENETUNREACH	ENOBUFS	ENODATA
ENOLINK	ENOPROTOOPT	ENOSR	ENOSTR
ENOTCONN	ENOTSOCK	EOPNOTSUPP	EOVERFLOW
EPROTO	EPROTONOSUPPORT	EPROTOTYPE	ESTALE
ETIME	ETIMEDOUT	EWOULDBLOCK	

5.6 Interprocess Communication (IPC)

The status of the *msg**( ) and *sem**( ) IPC functions in Issue 4 was changed from optional to mandatory. The *shm**( ) functions became part of the Shared Memory feature group. Thus all XSI-conforming systems provide the same mechanisms for manipulating messages and semaphores. However, X/Open is still considering the problem of generalised IPC, which is not addressed by the IPC functions defined in Issue 4.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. These interfaces are currently published in IEEE Std 1003.1b-1993 (POSIX.1b), and will be published in an amended POSIX-1 standard. Application developers are advised to write their code in such a way that modules using IPC interfaces described in Issue 4 can be modified easily in the future to use alternative interfaces.

5.7 STREAMS

The X/Open UNIX Extension in Issue 4, Version 2 introduces traditional System V STREAMS to the XSH specification. STREAMS are a method of implementing network services and other character-based input/output mechanisms, with the STREAM being a full-duplex connection between a process and a device. STREAMS provides direct access to protocol modules, and optional protocol modules can be interposed between the process-end of the STREAM and the device-driver at the device-end of the STREAM.

The XSH specification, Section 2.5, STREAMS, introduces X/Open UNIX STREAMS I/O, the message types used to control them, an overview of the priority mechanism, and the interfaces used to access them.

5.8 Makefile Portability

Makefiles are as much a part of the source-code to be ported, as the C-language files that implement the application's function. Every implementation of *make* has extended the syntax and added functionality to serve specific groups of users, and satisfy certain perceived shortcomings of the original utility. Some of the disparate features found include:

- syntax to support parallel (multi-processor) execution
- additional special targets
- additional operators
- support for preprocessor-like functionality to ''include'' other files, and conditionally process sections of the makefile
- changes in internal macro semantics when working with library targets
- support for remote execution.

The Single UNIX Specification defers to the POSIX.2 *make* utility. The POSIX.2 working group chose to describe the most common features found in System V and BSD variants of *make* to define the standard behaviour.

The special target *.POSIX* was defined such that if it appears prior to the first non-comment line in a makefile it ensures that the makefile is processed in the standard manner. This target does not prevent or preclude extended functions and features, but these extensions shall not alter the behaviour of the ''standard'' definition. The only extension to the POSIX.2 standard definition of *make* defined by the XCU specification is to allow for support for *sccs*. To use the extended *sccs*-based default rules, the special target *.SCCS_GET* must appear in the makefile.

An organisation may have a considerable amount of history and folklore embedded in their makefiles. Determining how old makefiles work can be a labour intensive part of porting an application. Investing the effort in the first ''port'' of the makefiles to the Single UNIX Specification definition of *make* means that they will be portable from that point forward to all platforms that are X/Open UNIX or XPG4 Base branded, or support POSIX.2.

A number of issues to be aware of with the standard *make*:

- The default macro for the C-compiler is *CC=c89*, and not *cc*. The *c89* compiler front-end invokes the standard C-language compiler.
- Although the Single UNIX Specification extends its specification of makefile syntax to support *sccs*, no support for RCS was added.
- Implementations support **MAKEFLAGS** with both the System V letter-string and the BSD command-line syntax, but they need not support both behaviours intermixed together.
- The historical **MAKESHELL** feature is not supported.
- The **–n** option specifies that commands should be written to standard output but not executed. However, it should be noted that commands with a plus sign (+) prefix will still be executed. The System V behaviour of still executing command-lines that contain the strings *$(MAKE)* or *${MAKE}* when a **–n** option is present is not supported.
- While there is existing *make* practice that allows the use of blanks to delimit command-lines, as well as the more traditional tab character, the standard *make* does not support this feature.
- New macros defined using other macros evaluate when the new macro is used, not when it was defined.

- The debug option (**−d**) is not specified although implementations will likely provide it in an implementation-specific manner. The output format of the **−p** option to see the complete presentation of macro definitions and target descriptions is not described.

Chapter 6

Interface Tables

This chapter contains a table of all the interfaces defined in Issue 4, Version 2, complete with an indication of their availability in Issue 3, the POSIX-1 standard, the POSIX-2 standard, the ISO C standard, SVID3 and 4.3BSD. The following conventions are used in columns 2 through 7:

m Indicates that the interface is defined as mandatory.

o Indicates that the interface is defined as optional.

f Indicates that the interface is part of a feature group; refer to Section 5.1 on page 92 and Section 1.7 on page 11 for a description of feature groups and under what conditions these interfaces are mandatory or optional.

U Indicates that the interface is part of the X/Open UNIX Extension feature group, new with Issue 4, Version 2.

† Indicates that the interface has been modified between Issue 4, and Issue 4, Version 2 for X/Open UNIX conformance. These changes are typically additional error conditions, although some modifications are quite extensive.

. Indicates that the interface is not specified.

The table is intended as a quick reference guide for programmers migrating or developing applications for Issue 4 conformance. Products that brand to the XPG4 UNIX profile provide all of the interfaces listed, regardless of whether or not they are part of a feature group. (The interfaces that are part of the Encryption feature group are the only possible exception to this, due to U.S. Government export restrictions on the decoding algorithm.)

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
a64l( )	.	U	.	.	m	.
abort( )	m	m	m	m	m	m
abs( )	m	m	m	m	m	m
access( )	m	m†	m	.	m	m
acos( )	m	m	m	m	m	m
acosh( )	.	U	.	.	m	m
advance( )	m	m	.	.	m	.
alarm( )	m	m†	m	.	m	m
asctime( )	m	m	m	m	m	m
asin( )	m	m	m	m	m	m
asinh( )	.	U	.	.	m	m
assert( )	m	m	m	m	m	m
atan( )	m	m	m	m	m	m
atan2( )	m	m	m	m	m	m
atanh( )	.	U	.	.	m	m
atexit( )	.	m†	.	m	m	.
atof( )	m	m	m	m	m	m
atoi( )	m	m	m	m	m	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
atol( )	m	m	m	m	m	m
basename( )	.	U	.	.	m	.
bcmp( )	.	U	.	.	.	m
bcopy( )	.	U	.	.	.	m
brk( )	.	U	.	.	m	m
bsd_signal( )	.	U	.	.	.	.
bsearch( )	m	m	m	m	m	.
bzero( )	.	U	.	.	.	m
calloc( )	m	m	m	m	m	m
catclose( )	m	m	.	.	m	.
catgets( )	m	m†	.	.	m	.
catopen( )	m	m†	.	.	m	.
cbrt( )	.	U	.	.	m	m
ceil( )	m	m	m	m	m	m
cfgetispeed( )	m	m	m	.	m	.
cfgetospeed( )	m	m	m	.	m	.
cfsetispeed( )	m	m†	m	.	m	.
cfsetospeed( )	m	m†	m	.	m	.
chdir( )	m	m†	m	.	m	m
chmod( )	m	m†	m	.	m	m
chown( )	m	m†	m	.	m	m
chroot( )	m	m†	.	.	.	m
clearerr( )	m	m	m	m	m	m
clock( )	m	m	.	m	m	.
close( )	m	m†	m	.	m	m
closedir( )	m	m	m	.	m	m
closelog( )	.	U	.	.	.	m
compile( )	m	m	.	.	m	.
confstr( )	.	m	m	.	.	.
cos( )	m	m	m	m	m	m
cosh( )	m	m	m	m	m	m
creat( )	m	m	m	.	m	m
crypt( )	o	f	.	.	m	m
ctermid( )	m	m	m	.	m	.
ctime( )	m	m	m	m	m	m
cuserid( )	m	m	.	.	m	.
daylight	m	m	.	.	.	.
dbm_clearerr( )	.	U	.	.	.	.
dbm_close( )	.	U	.	.	.	.
dbm_delete( )	.	U	.	.	.	.
dbm_error( )	.	U	.	.	.	.
dbm_fetch( )	.	U	.	.	.	.
dbm_firstkey( )	.	U	.	.	.	.
dbm_nextkey( )	.	U	.	.	.	.
dbm_open( )	.	U	.	.	.	.
dbm_store( )	.	U	.	.	.	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
difftime( )	.	m	.	m	m	.
dirname( )	.	U	.	.	.	.
div( )	.	m	.	m	m	.
drand48( )	m	m	.	.	m	.
dup( )	m	m	m	.	m	m
dup2( )	m	m	m	.	m	m
ecvt( )	.	U	.	.	.	m
encrypt( )	o	f	.	.	m	m
endgrent( )	.	U	.	.	m	m
endpwent( )	.	U	.	.	m	m
endutxent( )	.	U	.	.	.	.
environ	m	m	m	.	.	.
erand48( )	m	m	.	.	m	.
erf( )	m	m	.	.	m	m
erfc( )	m	m	.	.	m	m
errno	m	m	m	m	.	.
execl( )	m	m†	m	.	m	m
execv( )	m	m†	m	.	m	m
execle( )	m	m†	m	.	m	m
execve( )	m	m†	m	.	m	m
execlp( )	m	m†	m	.	m	m
execvp( )	m	m†	m	.	m	m
exit( )	m	m†	m	m	m	m
_exit( )	m	m†	m	m	m	m
exp( )	m	m	m	m	m	m
expm1( )	.	U	.	.	.	m
fabs( )	m	m	m	m	m	m
fattach( )	.	U	.	.	m	.
fchdir( )	.	U	.	.	m	.
fchmod( )	.	U	.	.	m	m
fchown( )	.	U	.	.	m	m
fclose( )	m	m	m	m	m	m
fcntl( )	m	m	m	.	m	m
fcvt( )	.	U	.	.	.	m
FD_CLR( )	.	U	.	.	.	.
FD_ISSET( )	.	U	.	.	.	.
FD_SET( )	.	U	.	.	.	.
FD_ZERO( )	.	U	.	.	.	.
fdetach( )	.	U	.	.	m	.
fdopen( )	m	m	m	.	m	m
feof( )	m	m	m	m	m	m
ferror( )	m	m	m	m	m	m
fflush( )	m	m	m	m	m	m
ffs( )	.	U	.	.	.	m
fgetc( )	m	m†	m	m	m	m
fgetpos( )	.	m	.	m	m	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
fgets( )	m	m	m	m	m	m
fgetwc( )	.	m†	.	.	.	.
fgetws( )	.	m	.	.	.	.
fileno( )	m	m	m	.	m	m
floor( )	m	m	m	m	m	m
fmod( )	m	m	m	m	m	.
fmtmsg( )	.	U	.	.	m	.
fnmatch( )	.	m	m	.	.	.
fopen( )	m	m†	m	m	m	m
fork( )	m	m†	m	.	m	m
fpathconf( )	m	m	m	.	m	.
fprintf( )	m	m†	m	m	m	m
fputc( )	m	m†	m	m	m	m
fputs( )	m	m	m	m	m	m
fputwc( )	.	m†	.	.	.	.
fputws( )	.	m	.	.	.	.
fread( )	m	m	m	m	m	m
free( )	m	m†	m	m	m	m
freopen( )	m	m†	m	m	m	m
frexp( )	m	m	m	m	m	m
fscanf( )	m	m	m	m	m	m
fseek( )	m	m†	m	m	m	m
fsetpos( )	.	m	.	m	m	.
fstat( )	m	m†	m	.	m	m
fstatvfs( )	.	U	.	.	m	.
fsync( )	m	m	.	.	m	m
ftell( )	m	m	m	m	m	m
ftime( )	.	U	.	.	.	m
ftok( )	.	U	.	.	.	.
ftruncate( )	.	U	.	.	.	.
ftw( )	m	m†	.	.	m	.
fwrite( )	m	m	m	m	m	m
gamma( )	m	m	.	.	m	.
gcvt( )	.	U	.	.	.	m
getc( )	m	m	m	m	m	m
getchar( )	m	m	m	m	m	m
getcontext( )	.	U	.	.	m	.
getcwd( )	m	m	m	.	m	.
getdate( )	.	U	.	.	m	.
getdtablesize( )	.	U	.	.	.	m
getegid( )	m	m	m	.	m	m
getenv( )	m	m	m	m	m	m
geteuid( )	m	m	m	.	m	m
getgid( )	m	m	m	.	m	m
getgrent( )	.	U	.	.	m	m
getgrgid( )	m	m	m	.	m	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
getgrnam( )	m	m	m	.	m	.
getgroups( )	m	m	m	.	m	m
gethostid( )	.	U	.	.	.	m
getitimer( )	.	U	.	.	m	m
getlogin( )	m	m	m	.	m	m
getmsg( )	.	U	.	.	m	.
getopt( )	m	m	m	.	m	m
getpagesize( )	.	U	.	.	.	m
getpass( )	m	m	.	.	m	m
getpgid( )	.	U	.	.	m	.
getpgrp( )	m	m	m	.	m	m
getpid( )	m	m	m	.	m	m
getpmsg( )	.	U	.	.	m	.
getppid( )	m	m	m	.	m	m
getpriority( )	.	U	.	.	.	m
getpwent( )	.	U	.	.	m	m
getpwnam( )	m	m	m	.	m	m
getpwuid( )	m	m	m	.	m	m
getrlimit( )	.	U	.	.	m	m
getrusage( )	.	U	.	.	.	m
gets( )	m	m	m	m	m	m
getsid( )	.	U	.	.	m	.
getsubopt( )	.	U	.	.	m	.
gettimeofday( )	.	U	.	.	m	m
getuid( )	m	m	m	.	m	m
getutxent( )	.	U	.	.	.	.
getutxid( )	.	U	.	.	.	.
getutxline( )	.	U	.	.	.	.
getw( )	m	m	.	.	m	m
getwc( )	.	m	.	.	.	.
getwchar( )	.	m	.	.	.	.
getwd( )	.	U	.	.	.	m
glob( )	.	f	m	.	.	.
globfree( )	.	f	m	.	.	.
gmtime( )	m	m	m	m	m	m
grantpt( )	.	U	.	.	m	.
hcreate( )	m	m	.	.	m	.
hdestroy( )	m	m	.	.	m	.
hsearch( )	m	m	.	.	.	.
hypot( )	m	m	.	.	m	m
iconv( )	.	m	.	.	.	.
iconv_close( )	.	m	.	.	.	.
iconv_open( )	.	m	.	.	.	.
ilogb( )	.	U	.	.	.	.
index( )	.	U	.	.	.	m
initstate( )	.	U	.	.	.	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
insque( )	.	U	.	.	.	m
ioctl( )	.	U	.	.	m	m
isalnum( )	m	m	m	m	m	m
isalpha( )	m	m	m	m	m	m
isascii( )	m	m	m	m	m	m
isastream( )	.	U	.	.	.	.
isatty( )	m	m	m	.	m	m
iscntrl( )	m	m	m	m	m	m
isdigit( )	m	m	m	m	m	m
isgraph( )	m	m	m	m	m	m
islower( )	m	m	m	m	m	m
isnan( )	m	m	.	.	m	.
isprint( )	m	m	m	m	m	m
ispunct( )	m	m	m	m	m	m
isspace( )	m	m	m	m	m	m
isupper( )	m	m	m	m	m	m
iswalnum( )	.	m	.	.	.	.
iswalpha( )	.	m	.	.	.	.
iswcntrl( )	.	m	.	.	.	.
iswctype( )	.	m	.	.	.	.
iswdigit( )	.	m	.	.	.	.
iswgraph( )	.	m	.	.	.	.
iswlower( )	.	m	.	.	.	.
iswprint( )	.	m	.	.	.	.
iswpunct( )	.	m	.	.	.	.
iswspace( )	.	m	.	.	.	.
iswupper( )	.	m	.	.	.	.
iswxdigit( )	.	m	.	.	.	.
isxdigit( )	m	m	m	m	m	m
j0( )	m	m	.	.	m	m
j1( )	m	m	.	.	m	m
jn( )	m	m	.	.	m	m
jrand48( )	m	m	.	.	m	.
kill( )	m	m	m	.	m	m
killpg( )	.	U	.	.	.	m
l64a( )	.	U	.	.	m	.
labs( )	.	m	.	m	m	.
lchown( )	.	U	.	.	m	.
lcong48( )	m	m	.	.	m	.
ldexp( )	m	m	m	m	m	m
ldiv( )	.	m	.	m	m	.
lfind( )	m	m	.	.	m	.
lgamma( )	m	m	.	.	m	m
link( )	m	m†	m	.	m	m
loc1	m	m	.	.	m	.
localeconv( )	.	m	.	m	m	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
localtime( )	m	m	m	m	m	m
lockf( )	.	U	.	.	m	.
locs	m	m	.	.	m	.
log( )	m	m	m	m	m	m
log10( )	m	m	m	m	m	m
log1p( )	.	U	.	.	.	m
logb( )	.	U	.	.	m	m
_longjmp( )	.	U	.	.	.	m
longjmp( )	m	m†	m	m	m	m
lrand48( )	m	m	.	.	m	.
lsearch( )	m	m	.	.	m	.
lseek( )	m	m	m	.	m	m
lstat( )	.	U	.	.	m	m
makecontext( )	.	U	.	.	.	.
malloc( )	m	m	m	m	m	m
mblen( )	.	m	.	m	m	.
mbstowcs( )	.	m	.	m	m	.
mbtowc( )	.	m	.	m	m	.
memccpy( )	m	m	.	.	m	.
memchr( )	m	m	.	m	m	.
memcmp( )	m	m	.	m	m	.
memcpy( )	m	m	.	m	m	.
memmove( )	.	m	.	m	m	.
memset( )	m	m	.	m	m	.
mkdir( )	m	m†	m	.	m	m
mkfifo( )	m	m†	m	.	m	.
mknod( )	.	U	.	.	m	m
mkstemp( )	.	U	.	.	.	m
mktemp( )	.	U	.	.	m	m
mktime( )	m	m	m	m	m	.
mmap( )	.	U	.	.	.	.
modf( )	m	m	m	m	m	m
mprotect( )	.	U	.	.	.	.
mrand48( )	m	m	.	.	m	.
msgctl( )	o	m	.	.	.	.
msgget( )	o	m	.	.	.	.
msgrcv( )	o	m	.	.	.	.
msgsnd( )	o	m	.	.	.	.
msync( )	.	U	.	.	.	.
munmap( )	.	U	.	.	.	.
nextafter( )	.	U	.	.	m	.
nftw( )	.	U	.	.	m	.
nice( )	m	m†	.	.	.	m
nl_langinfo( )	m	m	.	.	m	.
nrand48( )	m	m	.	.	m	.
open( )	m	m†	m	.	m	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
opendir( )	m	m†	m	.	m	m
openlog( )	.	U	.	.	.	m
optarg	m	m	m	.	m	m
pathconf( )	m	m†	m	.	m	.
pause( )	m	m	m	.	m	m
pclose( )	m	m	m	.	m	m
perror( )	m	m	m	m	m	m
pipe( )	m	m†	m	.	m	m
poll( )	.	U	.	.	m	.
popen( )	m	m	m	.	m	m
pow( )	m	m	m	m	m	m
printf( )	m	m	m	m	m	m
ptsname( )	.	U	.	.	m	.
putc( )	m	m	m	m	m	m
putchar( )	m	m	m	m	m	m
putenv( )	m	m	.	.	m	.
putmsg( )	.	U	.	.	m	.
putpmsg( )	.	U	.	.	m	.
puts( )	m	m	m	m	m	m
pututxline( )	.	U	.	.	.	.
putw( )	m	m	.	.	m	m
putwc( )	.	m	.	.	.	.
putwchar( )	.	m	.	.	.	.
qsort( )	m	m	m	m	m	m
raise( )	.	m	.	m	m	.
rand( )	m	m	m	m	m	m
random( )	.	U	.	.	.	m
re_comp( )	.	U	.	.	.	m
re_exec( )	.	U	.	.	.	m
read( )	m	m†	m	.	m	m
readdir( )	m	m†	m	.	m	m
readlink( )	.	U	.	.	m	m
readv( )	.	U	.	.	m	m
realloc( )	m	m	m	m	m	m
realpath( )	.	U	.	.	.	.
regcmp( )	.	U	.	.	.	.
regcomp( )	.	f	m	.	.	.
regerror( )	.	f	m	.	.	.
regex( )	.	U	.	.	.	m
regexec( )	.	f	m	.	.	.
regexp( )	m	m	.	.	.	.
regfree( )	.	f	m	.	.	.
remainder( )	.	U	.	.	m	.
remove( )	m	m	m	m	m	.
remque( )	.	U	.	.	.	m
rename( )	m	m†	m	m	m	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
rewind( )	m	m	m	m	m	m
rewinddir( )	m	m	m	.	m	m
rindex( )	.	U	.	.	.	m
rint( )	.	U	.	.	.	m
rmdir( )	m	m†	m	m	m	m
sbrk( )	.	U	.	.	m	m
scalb( )	.	U	.	.	m	m
scanf( )	m	m	m	m	m	m
seed48( )	m	m	.	.	m	.
seekdir( )	m	m†	.	.	m	m
select( )	.	U	.	.	.	m
semctl( )	o	m	.	.	.	.
semget( )	o	m	.	.	.	.
semop( )	o	m	.	.	.	.
setbuf( )	m	m	m	m	m	m
setcontext( )	.	U	.	.	m	.
setgid( )	m	m	m	.	m	m
setgrent( )	.	U	.	.	m	m
setitimer( )	.	U	.	.	m	m
_setjmp( )	.	U	.	.	.	m
setjmp( )	m	m	m	m	m	m
setkey( )	o	f	.	.	m	m
setlocale( )	m	m	m	m	m	.
setlogmask( )	.	U	.	.	.	m
setpgid( )	o	m	m	.	m	.
setpgrp( )	.	U	.	.	m	m
setpriority( )	.	U	.	.	.	m
setpwent( )	.	U	.	.	m	m
setregid( )	.	U	.	.	.	m
setreuid( )	.	U	.	.	.	m
setrlimit( )	.	U	.	.	m	m
setsid( )	m	m	m	.	m	.
setstate( )	.	U	.	.	.	m
setuid( )	m	m	m	.	m	m
setutxent( )	.	U	.	.	.	.
setvbuf( )	m	m	.	m	m	.
shmat( )	o	f	.	.	.	.
shmctl( )	o	f†	.	.	.	.
shmdt( )	o	f	.	.	.	.
shmget( )	o	f	.	.	.	.
sigaction( )	m	m†	m	.	m	.
sigaddset( )	m	m	m	.	m	.
sigaltstack( )	.	U	.	.	m	.
sigdelset( )	m	m	m	.	m	.
sigemptyset( )	m	m	m	.	m	.
sigfillset( )	m	m	m	.	m	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
sighold( )	.	U	.	.	m	.
sigignore( )	.	U	.	.	m	.
siginterrupt( )	.	U	.	.	.	m
sigismember( )	m	m	m	.	m	.
siglongjmp( )	m	m	m	.	m	.
signal( )	m	m†	.	m	m	m
signgam	m	m	.	.	.	.
sigpause( )	.	U	.	.	m	m
sigpending( )	m	m	m	.	m	.
sigprocmask( )	m	m	m	.	m	.
sigrelse( )	.	U	.	.	m	.
sigset( )	.	U	.	.	m	.
sigsetjmp( )	m	m	m	.	m	.
sigstack( )	.	U	.	.	.	m
sigsuspend( )	m	m	m	.	m	.
sin( )	m	m	m	m	m	m
sinh( )	m	m	m	m	m	m
sleep( )	m	m	m	.	m	m
sprintf( )	m	m	m	m	m	m
sqrt( )	m	m	m	m	m	m
srand( )	m	m	m	m	m	m
srand48( )	m	m	.	.	m	.
srandom( )	.	U	.	.	.	m
sscanf( )	m	m	m	m	m	m
stat( )	m	m†	m	.	m	m
statvfs( )	.	U	.	.	m	.
stderr	m	m	m	m	m	m
stdin	m	m	m	m	m	m
stdout	m	m	m	m	m	m
step( )	m	m	.	.	m	.
strcasecmp( )	.	U	.	.	.	.
strcat( )	m	m	m	m	m	m
strchr( )	m	m	m	m	m	.
strcmp( )	m	m	m	m	m	m
strcoll( )	m	m	.	m	m	.
strcpy( )	m	m	m	m	m	m
strcspn( )	m	m	m	m	m	.
strdup( )	.	U	.	.	m	.
strerror( )	m	m	.	m	m	.
strfmon( )	.	f	.	.	.	.
strftime( )	m	m	m	m	m	.
strlen( )	m	m	m	m	m	m
strncasecmp( )	.	U	.	.	.	.
strncat( )	m	m	m	m	m	m
strncmp( )	m	m	m	m	m	m
strncpy( )	m	m	m	m	m	m

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
strpbrk( )	m	m	m	m	m	.
strptime( )	.	f	.	.	.	.
strrchr( )	m	m	m	m	m	.
strspn( )	m	m	m	m	m	.
strstr( )	m	m	m	m	m	.
strtod( )	m	m	.	m	m	.
strtok( )	m	m	m	m	m	.
strtol( )	m	m	.	m	m	.
strtoul( )	.	m	.	m	m	.
strxfrm( )	m	m	.	m	m	.
swab( )	m	m	.	.	m	m
swapcontext( )	.	U	.	.	.	.
symlink( )	.	U	.	.	m	m
sync( )	.	U	.	.	m	m
sysconf( )	m	m	m	.	m	.
syslog( )	.	U	.	.	.	m
system( )	m	m	.	m	m	m
tan( )	m	m	m	m	m	m
tanh( )	m	m	m	m	m	m
tcdrain( )	m	m	m	.	m	.
tcflow( )	m	m	m	.	m	.
tcflush( )	m	m	m	.	m	.
tcgetattr( )	m	m	m	.	m	.
tcgetpgrp( )	o	m	m	.	m	.
tcgetsid( )	.	U	.	.	m	.
tcsendbreak( )	m	m	m	.	m	.
tcsetattr( )	m	m	m	.	m	.
tcsetpgrp( )	o	m	m	.	m	.
tdelete( )	m	m	.	.	m	.
telldir( )	m	m	.	.	m	m
tempnam( )	m	m	.	.	m	.
tfind( )	m	m	.	.	m	.
time( )	m	m	m	m	m	m
times( )	m	m	m	.	m	m
timezone	m	m	.	.	m	.
tmpfile( )	m	m	m	m	m	.
tmpnam( )	m	m	m	m	m	.
toascii( )	m	m	.	.	m	m
_tolower( )	m	m	.	.	m	.
tolower( )	m	m	m	m	m	m
_toupper( )	m	m	.	.	m	.
toupper( )	m	m	m	m	m	m
towlower( )	.	m	.	.	.	.
towupper( )	.	m	.	.	.	.
truncate( )	.	U	.	.	.	m
tsearch( )	m	m	.	.	m	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
ttyname( )	m	m	m	.	m	m
ttyslot( )	.	U	.	.	.	m
twalk( )	m	m	.	.	m	.
tzname	m	m	m	.	m	.
tzset( )	m	m	m	.	m	.
ualarm( )	.	U	.	.	.	m
ulimit( )	m	m	.	.	m	.
umask( )	m	m	m	.	m	m
uname( )	m	m	m	.	m	.
ungetc( )	m	m	m	m	m	m
ungetwc( )	.	m	.	.	.	.
unlink( )	m	m	m	.	m	m
unlockpt( )	.	U	.	.	m	.
usleep( )	.	U	.	.	.	m
utime( )	m	m†	m	.	m	m
utimes( )	.	U	.	.	.	m
valloc( )	.	U	.	.	.	m
vfork( )	.	U	.	.	.	m
vfprintf( )	m	m	.	.	m	.
vprintf( )	m	m	.	.	m	.
vsprintf( )	m	m	.	.	m	.
wait( )	m	m†	m	.	m	m
wait3( )	.	U	.	.	.	m
waitid( )	.	U	.	.	m	.
waitpid( )	m	m	m	.	m	m
wcscat( )	.	m	.	.	.	.
wcschr( )	.	m	.	.	.	.
wcscmp( )	.	m	.	.	.	.
wcscoll( )	.	f	.	.	.	.
wcscpy( )	.	m	.	.	.	.
wcscspn( )	.	m	.	.	.	.
wcsftime( )	.	f	.	.	.	.
wcslen( )	.	m	.	.	.	.
wcsncat( )	.	m	.	.	.	.
wcsncmp( )	.	m	.	.	.	.
wcsncpy( )	.	m	.	.	.	.
wcspbrk( )	.	m	.	.	.	.
wcsrchr( )	.	m	.	.	.	.
wcsspn( )	.	m	.	.	.	.
wcstod( )	.	m	.	.	.	.
wcstok( )	.	m	.	.	.	.
wcstol( )	.	m	.	.	.	.
wcstombs( )	.	m	.	m	m	.
wcstoul( )	.	m	.	.	.	.
wcswcs( )	.	m	.	.	.	.
wcswidth( )	.	m	.	.	.	.

Interface	Issue 3	Issue 4, Version 2	POSIX-1 and POSIX-2	ISO C	SVID3	4.3BSD
wcsxfrm( )	.	f	.	.	.	.
wctomb( )	.	m	.	m	m	.
wctype( )	.	m	.	.	.	.
wcwidth( )	.	m	.	.	.	.
wordexp( )	.	f	m	.	.	.
wordfree( )	.	f	m	.	.	.
write( )	m	m†	m	.	m	m
writev( )	.	U	.	.	m	m
y0( )	m	m	.	.	m	m
y1( )	m	m	.	.	m	m
yn( )	m	m	.	.	m	m

Chapter 7

System Interfaces

This chapter contains a section for each system interface defined in the XSH specification. Each section identifies syntax and semantic changes made to the interface in Issue 4 (if any), complete with examples where appropriate. Only changes that might affect an application programmer are identified.

As stated earlier, all interfaces in Issue 4 are defined in terms of function prototypes. This is a global comment on XSI Issue 4 and is not repeated individually in the following sections.

a64l( )

Issue 4, Version 2:

The conversion interfaces *a64l*( ) and *l64a*( ) were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Calls to *l64a*( ) may return a pointer to a static buffer, and therefore subsequent calls to *l64a*( ) may overwrite the buffer.

abort( )

Issue 4: The argument list is explicitly defined as **void**.

The **DESCRIPTION** section is updated to align precisely with the ISO C standard. In the main, this means describing capability that is already supported on Issue 3 systems, but is not formally specified in Issue 3. For example, the following are all defined:

- the order in which operations occur
- how the calling process is signalled
- how status information is made available to the host environment
- that the *abort*( ) function overrides blocking or ignoring of the SIGABRT signal.

abs( )

Issue 4: No functional changes are made, but developers should note that in the **APPLICATION USAGE** section, the phrase ‘‘{INT_MIN} is undefined’’ is replaced with ‘‘{INT_MIN} might not be representable’’.

access( )

Issue 4: The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link, and [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution.

acos( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] may be returned if the value of *x* is ±Inf or NaN.

Note: Whether or not a system supports Inf and NaN is implementation-dependent, meaning that portable applications should check for both zero and NaN as possible error return values.

acosh( )

Issue 4, Version 2:
The inverse hyperbolic functions *acosh*( ), *asinh*( ) and *atanh*( ) were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

advance( )

Issue 4: The header **<regexp.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of arguments *string* and *expbuf* are changed from **char*** to **const char***.

The interface is marked TO BE WITHDRAWN, because improved capability is now provided by interfaces introduced for alignment with the POSIX-2 standard.

alarm( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is included in the **SYNOPSIS** section.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated to indicate that interactions with the *setitimer*( ), *ualarm*( ) and *usleep*( ) functions are unspecified.

asctime( )

Issue 4: The type of argument *timeptr* is changed from **struct tm*** to **const struct tm***.

The location of the **tm** structure is now defined.

Writers of portable applications are advised to use the *strftime*( ) function rather than *asctime*( ). The *asctime*( ) function returns a string containing English language names and U.S. cultural conventions, whereas *strftime*( ) returns a string containing locale-dependent language and cultural values.

asin( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] may be returned if *x* is ±Inf or NaN.
- [ERANGE] may be returned if the result underflows.

Note: Whether or not a system supports Inf and NaN is implementation-dependent, meaning that portable applications should check for both zero and NaN as possible error return values.

asinh( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

assert( )

Issue 4: No changes are made in this issue, although note that the description of NDEBUG is moved from the **APPLICATION USAGE** section to the **DESCRIPTION** section.

atan( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if the result underflows.

Note: Whether or not a system supports Inf and NaN is implementation-dependent, meaning that portable applications should check for both zero and NaN as possible error return values.

atan2( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be set if the result underflows.

Note: Note that whether or not a system supports Inf and NaN is implementation-dependent, meaning that portable applications should check for both zero and NaN as possible error return values.

atanh( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

atexit( )

Issue 4: New function in Issue 4.

The *atexit*( ) function allows applications to specify tidy-up routines that are entered automatically by the system when the program terminates normally. These are entered in the reverse order of their registration.

Normal program termination is defined as a call to the *exit*( ) function or as a return from *main*( ). Thus, a program error that is not caught or ignored, or a call to *_exit*( ), bypasses this mechanism.

Issue 4, Version 2:
The **APPLICATION USAGE** section has been updated to indicate how the

application can determine {ATEXIT_MAX}, a constant added for X/OPEN UNIX conformance.

atof()

Issue 4: The type of argument *str* is changed from **char*** to **const char***.

Note: The capability provided by this interface is subsumed by the *strtod*() function. The *atof*() function is retained in the interface definition for compatibility with previous issues of the XSH specification, and because it is used extensively by existing applications. New applications should use the *strtod*() function, as it provides error checking of the input.

atoi()

Issue 4: The type of argument *str* is changed from **char*** to **const char***.

Note: The capability provided by this interface is subsumed by the *strtol*() function. The *atoi*() function is retained in the interface definition for compatibility with previous issues of the XSH specification, and because it is used extensively by existing applications. New applications should use the *strtol*() function, as it provides error checking of the input.

atol()

Issue 4: The type of argument *str* is changed from **char*** to **const char***.

Note: The capability provided by this interface is subsumed by the *strtol*() function. The *atol*() function is retained in the interface definition for compatibility with previous issues of the XSH specification, and because it is used extensively by existing applications. New applications should use the *strtol*() function, as it provides error checking of the input.

basename()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *basename*() function returns a pointer to the final component of a pathname pointed to by *path*, with trailing '/' characters removed. It may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten in a subsequent call to *basename*().

bcmp()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *bcmp*() function compares the first *n* bytes of the area pointed to by *s1* with the area pointed to by *s2*. For portability to implementations conforming to earlier versions of this document, *memcmp*() is preferred over this function.

bcopy()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *bcopy*( ) function copies *n* bytes from the area pointed to by *s1* to the area pointed to by *s2*. For portability to implementations conforming to earlier versions of this document, *memmove*( ) is preferred over this function.

brk()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *brk*( ) and *sbrk*( ) functions are used to change the amount of space allocated for the calling process. The behaviour of these functions is unspecified if the application also uses any of the other memory functions (for example *malloc*( ), *mmap*( ) or *free*( )). The *brk*( ) and *sbrk*( ) functions have been used in specialised cases where no other memory allocation function provided the same capability. Use of the *mmap*( ) function is now preferred as it can be used portably with all other memory allocation functions and with any function that uses other allocation functions.

bsd_signal()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

This function is a direct replacement for the BSD *signal*( ) function for simple applications that are installing a single-argument signal handler function. If a BSD signal handler function is being installed that expects more than one argument, the application has to be modified to use *sigaction*( ). The *bsd_signal*( ) function differs from *signal*( ) in that the SA_RESTART flag is set and the SA_RESETHAND will be clear when *bsd_signal*( ) is used. The state of these flags is not specified for *signal*( ).

bsearch()

Issue 4: The type of arguments *key* and *base*, and the type of arguments to the *compar*( ) function, are changed from **void*** to **const void***.

The requirement that the table be sorted according to *compar*( ) is no longer stated in the **DESCRIPTION**. This change should not affect existing applications and is made for alignment with the ISO C standard.

bzero()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *bzero*( ) function places *n* zero-valued bytes in the area pointed to by *s*. For portability to implementations conforming to earlier versions of this document, *memset*( ) is preferred over this function.

calloc( )

Issue 4: The description is updated to align with the ISO C standard. These changes are largely superficial, although they should be noted as they may have a marginal affect on some applications. Specifically:

- The order and contiguity of storage allocated by successive calls to this function is unspecified.
- Each allocation yields a pointer to an object disjoint from any other object.
- The returned pointer points to the lowest byte address of the allocation.
- The behaviour is implementation-dependent if the space requested is zero size, although the value returned must be either a null pointer or a unique pointer.
- A null pointer or a unique pointer is also returned if either *nelem* or *elsize* is zero.

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

catclose( )

Issue 4: Issue 4 defines that *catclose*( ) may return [EBADF] or [EINTR]. No errors are defined in Issue 3.

catgets( )

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

Issue 4 defines that *catclose*( ) may return [EBADF] or [EINTR]. No errors are defined in Issue 3.

Issue 4, Version 2:
The **RETURN VALUE** section notes that *errno* may be set to indicate an error, and the **ERRORS** section was updated to include the optional errors [EINVAL] and [ENOMSG].

catopen( )

Issue 4: The type of argument *name* is changed from **char*** to **const char***.

The description is updated to indicate:

- the longevity of message catalogue descriptors; that is, they remain valid in a process until closed or until one of the *exec* functions is called successfully
- the effects of *oflag* argument and the effects of the LC_MESSAGES category on *NLSPATH* substitutions.

Issue 3 defines that *NLSPATH* substitutions are performed using the settings of the *LANG* environment variable. Issue 4 supports this definition if the value of the *oflag* argument is 0. If *oflag* is set to NL_CAT_LOCALE, the setting of the LC_MESSAGES locale category is used for *NLSPATH* substitutions instead. This feature is new in Issue 4.

Issue 4 defines that *catopen*( ) may return [EACCES], [EMFILE], [ENAMETOOLONG], [ENFILE], [ENOENT], [ENOMEM] or [ENOTDIR]. Only [ENOMEM] is defined in Issue 3.

Note: Message catalogues may not be valid after a call to one of the *exec* functions. Note also that at present there are no guidelines for the location of message catalogues. Make sure names are chosen that do not conflict with catalogues used by other functions and the standard utilities.

Issue 4, Version 2:

For X/OPEN UNIX conformance, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

cbrt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX). The *cbrt*( ) function calculates the cube root of *x*.

ceil( )

Issue 4:

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- In Issue 4 [ERANGE] is set if the result overflows. In Issue 3 detection of this condition is optional.

cfgetispeed( )

Issue 4:

The type of argument *termios_p* is changed from **struct termios*** to **const struct termios***.

Issue 4 states that the function returns exactly the value in the **termios** structure. Issue 3 indicates that if this value is not obtained by a previous successful call to *tcgetattr*( ), the behaviour is undefined.

cfgetospeed( )

Issue 4:

The type of argument *termios_p* is changed from **struct termios*** to **const struct termios***.

Issue 4 states that the function returns exactly the value in the **termios** structure. Issue 3 indicates that if this value is not obtained by a previous successful call to *tcgetattr*( ), the behaviour is undefined.

cfsetispeed( )

Issue 4:

No changes are made to this interface, although note that the setting of *errno* and the [EINVAL] error are marked as extensions. This is done because these conditions are not specified in the POSIX-1 standard.

Issue 4, Version 2:

The **ERRORS** section has been updated to indicate that a second [EINVAL] condition may be detected if the speed given is outside the range of possible speed values given in **<termios.h>**.

cfsetospeed( )

Issue 4: No changes are made to this interface, although note that the setting of *errno* and the [EINVAL] error are marked as extensions. This is done because these conditions are not specified in the POSIX-1 standard.

Issue 4, Version 2:
The **ERRORS** section has been updated to indicate that a second [EINVAL] condition may be detected if the speed given is outside the range of possible speed values given in **<termios.h>**.

chdir( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:
For X/OPEN UNIX conformance, a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link, and [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution.

chmod( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included. Also note that any program that does include this header still compiles and runs correctly.

The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:
The **DESCRIPTION** section was updated to describe X/OPEN UNIX functionality relating to permission checks applied when removing or renaming files in a directory having the S_ISVTX bit set.

For X/OPEN UNIX conformance, [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link. The error return [EINTR] also may be returned if a signal was caught during execution of the function.

chown( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included. Also note that any program which does include this header still compiles and runs correctly.

The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, pathname can be truncated by the system, so the condition is silently ignored.

Issue 4 defines that [EPERM] is always returned when an attempt is made to change the user ID of a file and the caller does not have appropriate privilege. Detection of this condition is optional in Issue 3.

The value **(uid_t)**–1 for *owner* is added to the interface definition, to allow the owner of a file to change the group ID only. This is a new extension.

Issue 4, Version 2:

For X/OPEN UNIX conformance, [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link. The error returns [EINTR] and [EIO] also may be returned if a signal was caught during execution of the function, or an I/O error occurred while reading or writing the filesystem.

chroot( )

Issue 4: The interface is marked TO BE WITHDRAWN, as there is no portable use that an application can make of it. Application developers are therefore advised not to make any use of this interface, or assume its existence on any XSI-conformant system, as it will disappear from subsequent issues of the XSH specification.

Issue 4, Version 2:

For X/OPEN UNIX conformance, [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

clearerr( )

Issue 4: No changes are made to this interface in Issue 4.

clock( )

Issue 4: The header **<time.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The description of this function in Issue 4, though functionally equivalent to Issue 3, is rewritten for clarity and consistency with the ISO C standard. Issue 4 also defines that **(clock_t)**–1 is returned if the processor time used is not available, or if its value cannot be represented.

Issue 4 defines that the value returned by this function is calibrated in units of CLOCKS_PER_SECOND, which is a constant defined in **<time.h>**. Issue 3 states that the return value is CPU time in microseconds. However, as CLOCKS_PER_SECOND is defined to be 1 million on all XSI-conformant systems, portable applications should not be affected by this change.

close( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the **DESCRIPTION** section was updated to describe the actions of closing a file descriptor that refers to a STREAMS-based file or either side of a pseudo-terminal.

Also, the error return [EIO] may be returned if an I/O error occurred while reading or writing the filesystem.

closedir( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included. Also note that any program which does include this header still compiles and runs correctly.

closelog( )

Issue 4, Version 2:
The interfaces that control a system logging facility, *closelog*( ), *openlog*( ), *setlogmask*( ) and *syslog*( ), were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

compile( )

Issue 4: The interface is marked TO BE WITHDRAWN, because improved capability is now provided by interfaces introduced for alignment with the POSIX-2 standard (see *regcomp*( )).

The header **<regexp.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *endbuf* is changed from **char*** to **const char***.

confstr( )

Issue 4: New function in Issue 4.

The *confstr*( ) function allows applications to obtain configuration-defined string values, for example:

_CS_PATH This gives a value for the *PATH* environment variable that finds all standard utilities.

Other values may be supported by specific implementations.

This interface is similar to *sysconf*( ), except that it is used where string values rather than numeric values are returned.

cos( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] may be set if *x* is NaN or ±Inf.
- [ERANGE] may be set if the result underflows.

cosh( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] is set if the result overflows. In Issue 3 detection of this condition is defined as optional.

creat( )

Issue 4: The headers **<sys/types.h>** and **<sys/stat.h>** are no longer required explicitly. These headers are optional on XSI-conformant systems, although the POSIX-1 standard shows these headers to be included. Also note that any program which does include these headers still compiles and runs correctly.

The type of argument *path* is changed from **char*** to **const char***.

crypt( )

Issue 4: In Issue 4 this function is part of the Encryption feature group.

The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of arguments *key* and *salt* are changed from **char*** to **const char***.

Issue 4 explicitly defines the characters that can be used in the *salt* argument, which are locale-independent. The definition given in Issue 3 is ambiguous, in that it defines these characters in terms of RE ranges (which might in theory vary from one locale to another). This is a clarification of the interface definition and does not imply any functional change.

ctermid( )

Issue 4: No changes are made to this interface, although it is rewritten for alignment with the POSIX-1 standard.

ctime( )

Issue 4: The type of argument *clock* is changed from **time_t*** to **const time_t***.

Writers of portable applications are advised to use the *strftime*( ) function rather than *ctime*( ). The *ctime*( ) function returns a string containing English language names and U.S. cultural conventions, whereas *strftime*( ) returns a string containing locale-dependent language and cultural values.

cuserid( )

Issue 4: The interface is marked TO BE WITHDRAWN, because of differences between the historical definition of this interface and the definition published in the IEEE Std 1003.1-1988 (POSIX.1) (and hence Issue 3). The interface is also removed from the POSIX-1 standard.

The description is changed to indicate that an implementation can determine the name returned by the function from the real or effective user ID of the process. Hence the anomalous behaviour described above.

As indicated in the XSH specification, Issue 2 capability can be obtained by using:

```
getpwuid(getuid());
```

Issue 3 capability can be obtained by using:

```
getpwuid(geteuid());
```

daylight

Issue 4: No changes are made to this interface in Issue 4.

dbm_clearerr( )

Issue 4, Version 2:
The *dbm_* database functions were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The set of functions create, access and modify a database. They are most useful for fast lookup of relatively static information that is indexed by a single key. They do not provide for multiple search keys per entry, they do not protect against multi-user access, nor do they provide many of the other useful functions associated with a more robust database management system.

dbm_close( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_delete( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_error( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_fetch( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_firstkey( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_nextkey( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_open( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

dbm_store( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

difftime( )

Issue 4: New function in Issue 4.

This interface calculates the difference between two calendar times and returns the result as a type **double**. The input times are values of type *time_t*, as returned, for example, by the *time*( ) function.

dirname( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

It returns a pointer to a string that is the pathname of the parent directory of the pathname pointed to by *path*. The *dirname*( ) function may modify the string pointed to by *path*, and may return a pointer to static storage that may then be overwritten in a subsequent call to *dirname*( ).

div()

Issue 4: New function in Issue 4.

This function divides one integer by another and returns a structure containing the quotient and remainder. For example:

```
...
int     a, b;
div_t   result;
...
result = div (a, b);
printf ("quotient = %d, remainder = %d\n",
    result.quot, result.rem);
...
```

drand48()

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Also in the **SYNOPSIS** section, **long** is replaced by **long int** and **unsigned short** is replaced by **unsigned short int**.

The description is changed to allow for implementations on which {LONG_BIT} is larger than 32 bits.

dup()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Text is added to the description indicating that the return value from *dup2*( ) is *fildes2* on successful completion, or –1 on failure. No functional change is implied by this added text.

The list of errors is revised to describe the values returned by *dup*( ) and *dup2*( ) separately. Again, this is intended for clarification and does not imply any change in capability.

ecvt( )

Issue 4, Version 2:

The *ecvt*( ), *fcvt*( ) and *gcvt*( ), functions that convert floating point-numbers to strings were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ecvt*( ) and *fcvt*( ) functions may return a pointer to static storage that may then be overwritten in subsequent calls to these functions.

For portability to implementations conforming to earlier versions of this document, *sprintf*( ) is preferred over this function.

encrypt( )

Issue 4: In Issue 4 this function is part of the Encryption feature group.

The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The description in Issue 4 states that the encoding algorithm for encryption is implementation-dependent. More importantly, it states that decoding need not be supported on all implementations; this is related to U.S. government restrictions about DES decryption algorithms being exported outside the U.S.A.

In Issue 4 this function is part of the Encryption feature group.

environ

Issue 4: No changes are made to this interface in Issue 4.

endgrent( )

Issue 4, Version 2:

The group database entry functions, *endgrent*( ), *getgrent*( ) and *setgrent*( ), were first introduced in Issue 4, Version 2 (X / OPEN UNIX).

The return value of *getgrent*( ) may point to a static area which is overwritten by a subsequent call to *getgrent*( ), as well as *getgrgid*( ) or *getgrnam*( ).

Applications should avoid dependencies on fields in the group database, regardless of how and where the group database is organised. These functions are provided due to their historical usage. For portability to implementations conforming to earlier versions of this document, and to avoid these dependencies, *getgrnam*( ) and *getgrgid*( ) are preferred.

endpwent( )

Issue 4, Version 2:

The user database entry functions, *endpwent*( ), *getpwent*( ) and *setpwent*( ), were first introduced in Issue 4, Version 2 (X / OPEN UNIX).

The return value for *getpwent*( ) may point to a static area which is overwritten by a subsequent call to *getpwent*( ), as well as *getpwuid*( ) or *getpwnam*( ).

Applications should avoid dependencies on fields in the password database, regardless of how and where it is organised. These functions are provided due to their historical usage. For portability to implementations conforming to earlier versions of this document, and to avoid these dependencies, *getpwuid*( ) is preferred.

endutxent( )

Issue 4, Version 2:

The user accounting database functions *endutxent*( ), *getutxent*( ), *getutxid*( ), *getutxline*( ), *pututxline*( ) and *setutxent*( ), were first introduced in Issue 4, Version 2 (X / OPEN UNIX).

The functions may return pointers to static storage, or may cache data, so application developers should pay careful attention to the **APPLICATION USAGE** section.

erand48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Also in the **SYNOPSIS** section, **unsigned short** is replaced by **unsigned short int**.

erf( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if the result underflows.

errno

Issue 4: Issue 4 guarantees that *errno* is set to zero at program startup, and that it is never reset to zero by any XSI function. This is the case on Issue 3 systems, but it is not stated explicitly in the interface definition.

Issue 4 is aligned with the ISO C standard, which permits *errno* to be a macro.

exec( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The **const** keyword is added to identifiers of constant type (for example, *path* and *file*).

Issue 4 defines that directory streams are closed in the new process image. The state of these streams is undefined in previous issues.

Conversion descriptors are new in Issue 4 and have implications for various system interfaces, including the *exec* family of functions. The state of conversion descriptors (and message catalogue descriptors) in a new process image is undefined. Thus, portable applications should not rely on their continued availability outside the context of the process that opened them.

The description of *exec* is also updated to describe interactions with another new function *atexit*( ). In this case, the interface definition states that any previously registered functions are no longer registered after a successful call to one of the *exec* functions. The reasons for this are self-evident.

Shared memory is no longer optional.

Various changes are made to the **ERRORS** section, as follows:

- The description of the [ENOEXEC] error has been changed to indicate that this error does not apply to the *execlp*( ) and *execvp*( ) functions.
- [ENOMEM] is returned if there is not enough memory to create the new process.
- Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to reflect resources not preserved across the *exec*( ) call (alternate signal stacks, memory mappings established through *mmap*( )) and those that are (resource limits, controlling terminal, interval timers). The effects of ST_NOSUID being set for a file system are also defined.

For X/OPEN UNIX conformance, [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

exit( )

Issue 4:

The header **<unistd.h>**, which contains the function prototype for the *_exit*( ) function, is added to the **SYNOPSIS** section.

Various changes are made to define interactions with the *atexit*( ) function. Specifically:

- Functions registered by *atexit*( ) are called in the reverse order of their registration, as many times as they are registered.
- If a function registered by *atexit*( ) fails to return, any remaining functions registered by *atexit*( ) are not called and the rest of *exit*( ) processing is not completed. This can happen, for example, if the function calls *_exit*( ) or *abort*( ), or if it receives a signal that is not caught or ignored.
- If *exit*( ) is called more than once (for example, from within a function registered by *atexit*( )), the results are undefined.

Issue 4 defines job control as mandatory for all conforming implementations. Issue 3 defines this capability as optional. The consequence for this interface is that if a call to *exit*( ) causes a process group to become orphaned, and if any member of the newly orphaned process group is stopped, a SIGHUP signal followed by a SIGCONT signal is sent to the newly orphaned process group.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to add appropriate references for interactions with *wait3*( ) and *waitid*( ), to describe interactions when SA_NOCLDWAIT is set or SIGCHLD is set to SIG_IGN, and to describe how each mapped memory object is unmapped.

exp( )

Issue 4:

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- Issue 4 states that [EDOM] may be set if the value of *x* is NaN. Issue 3 defined the detection of this condition as mandatory.
- Similarly, Issue 4 states that [ERANGE] may be returned if the result underflows. Again, Issue 3 defined the detection of this condition as mandatory.

expm1( )

Issue 4, Version 2:

This function was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *expm1*( ) function computes exponential functions. The **APPLICATION USAGE** section describes that the value of *expm1(x)* may be more accurate than *exp(x)*–1.0 for small values of *x*, and that *expm1*( ) and *log1p*( ) are useful for certain financial calculations.

fabs( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if the result underflows.

fattach( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fattach*( ) function attaches a STREAMS-based file descriptor to a file. A successful call to *fattach*( ) causes all pathnames that name the file named by *path* to name the STREAMS file associated with *fildes*, until the STREAMS file is detached from the file. A STREAMS file can be attached to more than one file and have several pathnames associated with it.

fchdir( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fchdir*( ) function behaves the same as *chdir*( ), except that the directory that is to be the new working directory is specified by the file descriptor *fildes*.

fchmod( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fchmod*( ) function behaves the same as *chmod*( ), except that the file whose permissions are to be changed is specified by the file descriptor *fildes*.

fchown( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fchdir*( ) function behaves the same as *chdir*( ), except that the file whose group and owner are to be changed is specified by the file descriptor *fildes*.

fclose( )

Issue 4: The description is changed to indicate that *fclose*( ) performs a *close*( ) on the underlying file descriptor. Issue 3 states that it called *fclose*( ) to do this, which is a documentation error in Issue 3.

The following paragraph is withdrawn (by POSIX as well as X/Open) because of the possibility of causing applications to malfunction, and the impossibility of implementing these mechanisms for pipes:

> ''If the file is not already at EOF, and the file is one capable of seeking, the file *offset* of the underlying open file description is adjusted so that the next operation on the open file description deals with the byte after the last one read from, or written to, the stream being closed.''

It is replaced with a statement that any subsequent use of *stream* is undefined.

Issue 4, Version 2:
The cross-reference to *getrlimit*( ) was added to the **SEE ALSO** section.

fcntl( )

Issue 4: The headers **<sys/types.h>** and **<unistd.h>** are no longer required explicitly. These headers are optional on XSI-conformant systems, although the POSIX-1 standard shows these headers to be included.

The meaning of a successful F_SETLK or F_SETLKW request is clarified, after a POSIX Request for Interpretation. This should not affect the operation of existing applications that conform to Issue 3.

fcvt( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

FD_CLR( )

Issue 4, Version 2:
The macros *FD_CLR*( ), *FD_ISSET*( ), *FD_SET*( ) and *FD_ZERO*( ) for synchronous I/O multiplexing (see *select*( )) were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

fdetach( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fdetach*( ) function detaches a STREAMS-based file from the file to which it was attached by a previous call to *fattach*( ).

fdopen()

Issue 4: The type of argument *mode* is changed from **char*** to **const char***.

The use and settings of the *mode* argument are changed to include binary streams. This change is made for alignment with the ISO C standard and should not affect applications that conform to Issue 3, as these flags are additive rather than replacements for Issue 3 flags. Note also that the b flag has no effect on an Issue 4 system.

feof()

Issue 4: No errors are defined for this interface in Issue 4. Issue 3 defines that [EBADF] can be returned. However, as detection of this condition is not mandated in Issue 3, it should not be relied on by conforming applications.

ferror()

Issue 4: No errors are defined for this interface in Issue 4. Issue 3 defines that [EBADF] can be returned, but as detection of this error is not mandated, it should not affect application portability.

fflush()

Issue 4: Issue 4 defines that if *stream* is a null pointer, the *fflush*( ) function performs flushing actions on all output streams and update streams (on which the most recent operation was not input). This feature of *fflush*( ) is not defined in Issue 3.

The following two paragraphs are withdrawn (by POSIX as well as X/Open) because of the possibility of causing applications to malfunction, and the impossibility of implementing these mechanisms for pipes:

> ''If the stream is open for reading, any unread data buffered in the stream is discarded.
>
> For a stream open for reading, if the file is not already at EOF, and the file is one capable of seeking, the file offset of the underlying open file description is adjusted so that the next operation on the open file description deals with the byte after the last one read from, or written to, the stream being flushed.''

ffs()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ff*s( ) function finds the first bit set (beginning at the least significant bit) and returns the index of that bit.

fgetc()

Issue 4: The description in Issue 4 is updated to make it clear that the function returns a byte value. Issue 3 states that a character is returned, which can be misinterpreted to mean a multi-byte sequence in locales that support multi-byte codesets.

The WPI functions *fgetwc*( ) and *fgetws*( ) are added to the list of functions that may cause the *st_atime* field to be updated.

Issue 4 defines that error returns are generated when either the stream is unbuffered, or data needs to be read into the stream buffer. This is a minor clarification and should not affect application portability.

In Issue 3, generation of the [EIO] error depends on whether or not an implementation supports job control. This capability is defined as mandatory in Issue 4.

A note is added to the **APPLICATION USAGE** section in Issue 4, indicating how an application can distinguish between an error condition and an end-of-file condition.

Issue 4, Version 2:

The **ERRORS** section has been updated to include the condition where a physical I/O error has occurred in the description of [EIO] for conformance to X/OPEN UNIX.

fgetpos( )

Issue 4: New function in Issue 4.

The *fgetpos*( ) function stores the current value of the file position indicator for the specified *stream* into an object of type **fpos_t**. The interface is similar to *ftell*( ), except that the format of the stored information is unspecified; that is, it is not necessarily a byte offset. The stream can be repositioned by calling *fsetpos*( ), passing the value of the file position indicator returned by a previous call to *fgetpos*( ).

fgets( )

Issue 4: The description in Issue 4 is updated to make it clear that the function reads bytes. Issue 3 states that characters are read, which can be misinterpreted to mean multi-byte sequences in locales that support multi-byte codesets.

The WPI functions *fgetwc*( ) and *fgetws*( ) are added to the list of functions that may cause the *st_atime* field to be updated.

fgetwc( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *fgetc*( ), except that it returns a wide-character code (rather than a byte). The return value is converted to type **wint_t**, which contains either the wide-character code if the call is successful, or the constant WEOF.

Issue 4, Version 2:

The **ERRORS** section has been updated to include the condition where a physical I/O error has occurred in the description of [EIO] for conformance to X/OPEN UNIX.

fgetws( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *fgets*( ), except that it returns a wide-character string. Multi-byte sequences are automatically converted to values of type **wchar_t** and stored in the array pointed to by *ws*.

fileno( )

Issue 4: No changes are made to this interface in Issue 4, although note that detection of the [EBADF] error is marked as an X/Open extension.

floor( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] is set if the result would cause overflow. In Issue 3 detection of this condition is not guaranteed.

fmod( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] may also be set if *x* is ±Inf. This is not specified in Issue 3.
- [ERANGE] may be set if the result would underflow.

fmtmsg( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *fmtmsg*( ) function displays a message in the specified format on standard error and/or a system console.

fnmatch( )

Issue 4: New function in the POSIX.2 C-language Binding feature group in Issue 4.

This function matches a filename or pathname against a specified pattern, returning zero for success, FNM_NOMATCH if there is no match, or another non-zero value if an error occurs.

The *fnmatch*( ) function allows shell-like filename pattern matching to be performed from the program level.

For example:

```
#include <stdio.h>
#include <fnmatch.h>

main (int argc, char **argv)
{
    if (argc != 3) {
         printf ("usage: fnmatch pattern string\n");
         exit (1);
    }

    if (fnmatch (argv[1], argv[2], 0) == 0)
        printf ("Match found\n");
    else
        printf ("No match\n");

    exit (0);
}
```

which if called with the arguments:

```
fnmatch '*.c' foo.c
```

would display ''Match found'', and if called with the arguments:

```
fnmatch '?.c' foo.c
```

would display ''No match''.

The third argument to the function call is a flag, which if set to FNM_PATHNAME forces slash characters in *string* to be matched explicitly.

fopen()

Issue 4: The type of arguments *filename* and *mode* are changed from **char*** to **const char***.

The use and settings of the *mode* argument are changed to include binary streams. This change is made for alignment with the ISO C standard and should not affect applications that are compliant with Issue 3, as these flags are additive rather than replacements for Issue 3 flags. Note also that the b flag has no effect on an Issue 4 system.

The *fsetpos*() function is added to the list of file positioning functions. This interface is not published in Issue 3.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

[EMFILE] may be set if {FOPEN_MAX} streams are currently open in the calling process. This error is not defined in Issue 3.

Issue 4, Version 2:

For X/OPEN UNIX conformance, [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

fork()

Issue 4: The argument list is explicitly defined as **void**.

The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included.

Though functionally identical to Issue 3, the description of this interface in Issue 4 is reorganised to improve clarity and to align more closely with the POSIX-1 standard.

The description of the [EAGAIN] error is updated to indicate that this error can also be returned if a system lacks the resources to create another process.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to identify that interval timers are reset in the child process.

fpathconf()

Issue 4: No changes are made to this interface in Issue 4, although note that the function's return value is now defined in full as **long int**.

fprintf()

Issue 4: The type of the *format* argument is changed from **char*** to **const char***.

The **DESCRIPTION** section is reworded or presented differently in a number of places for alignment with the ISO C standard, and also for clarity. There are no functional changes, except as noted elsewhere in this section.

The C and S conversion characters are added, indicating respectively a wide character of type **wchar_t** and pointer to a wide-character string of type **wchar_t*** in the argument list.

The ' (single-quote) flag is added to the list of flag characters and marked as an extension. This flag directs that the integer portion of the result of a decimal conversion is formatted with the thousands grouping character.

Issue 4, Version 2:
The **ERRORS** section has been updated for X/OPEN UNIX conformance to add the optional error [ENOMEM] for *printf*( ) and *fprintf*( ).

fputc()

Issue 4: The description in Issue 4 is updated to make it clear that the function writes a byte value. Issue 3 states that a character is written, which can be misinterpreted to mean a multi-byte sequence in locales that support multi-byte codesets.

Issue 4 states that error returns are only generated either when the stream is unbuffered, or if the stream buffer needs to be flushed. Though implicit in Issue 3, this behaviour is not stated explicitly.

Issue 4 mandates that the [EIO] error is returned if the calling process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. Detection of this condition in Issue 3 depends on whether an implementation supports job control.

Issue 4 states that [ENXIO] may be returned if a request is made to a non-existent device, or if the request is outside the capabilities of the device. Detection of this condition is mandated in Issue 3.

Issue 4, Version 2:

The **ERRORS** section has been updated to include the condition where a physical I/O error has occurred in the description of [EIO] for conformance to X/OPEN UNIX.

fputs()

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

The words ''null character'' in the description are replaced by ''null byte'', to make it clear that this interface deals solely in byte values.

fputwc()

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *fputc*(), except that it accepts a wide-character code as input. This is converted to a (possibly) multi-byte sequence before writing.

Issue 4, Version 2:

The **ERRORS** section has been updated to include the condition where a physical I/O error has occurred in the description of [EIO] for conformance to X/OPEN UNIX.

fputws()

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *fputs*, except that it accepts a wide-character string as input. Before writing, this is converted to a byte sequence, which may contain multi-byte character codes (if defined in the current locale).

fread()

Issue 4: Issue 4 defines that if *size* or *nitems* is zero, *fwrite*() returns zero and the contents of the array and the state of the stream remain unchanged. This capability is not defined in Issue 3.

The *fgetwc*() and *fgetws*() functions are added to the list of functions that may cause the *st_atime* field to be updated.

free()

Issue 4: Issue 4 states that the consequences of using a pointer that refers to freed space are undefined. This is not stated explicitly in Issue 3, though it should be self-evident.

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate that *free*() can be used to free memory allocated by *valloc*().

freopen()

Issue 4: The type of arguments *filename* and *mode* are changed from **char*** to **const char***.

In Issue 4, the term ‘‘name’’ is replaced by ‘‘pathname’’, to make it clear that the interface is not limited to accepting filenames only.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

The description of the [EMFILE] error has been changed to refer to {OPEN_MAX} file descriptors rather than {FOPEN_MAX} file descriptors, directories and message catalogues. This is an error in Issue 3.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

frexp()

Issue 4: The name of the first argument is changed from *value* to *num*.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- Issue 4 states that if *num* is ±Inf, *num* is returned and *errno* may be set to [EDOM]]. Issue 3 defines that in these circumstances, an implementation-dependent value is returned.

fscanf()

Issue 4: The type of argument *format* for all functions, and the type of argument *s* for the *sscanf*( ) function, are changed from **char*** to **const char***.

The description is updated in various places to align more closely with the text of the ISO C standard. In particular, Issue 4 fully defines the L conversion character, allows for the support of multi-byte coded character sets (although these are not mandated by X/Open), and fills in a number of gaps in the definition (for example, by defining termination conditions for the *sscanf*( ) function).

Following an ANSI interpretation, the effect of conversion specifications that consume no input is better defined, and is no longer marked as an extension.

The C and S conversion characters are added, indicating a pointer in the argument list to the initial wide-character code of an array large enough to accept the input sequence.

Use of the terms ‘‘byte’’ and ‘‘character’’ is rationalised to make it clear when single-byte and multi-byte values can be used. Similarly, use of the terms ‘‘conversion specification’’ and ‘‘conversion character’’ is more precise.

Various errors are corrected. For example, the description of the d conversion character contains an erroneous reference to the *strtod*( ) function in Issue 3. This is

replaced in Issue 4 by reference to the *strtol*( ) function.

The description is updated in a number of places to indicate further implications of the %*n*$ form of a conversion. All references to this capability, which is not specified in the ISO C standard, are marked as extensions.

The **ERRORS** section is changed to refer to the entries for *fgetc*( ) and *fgetwc*( ), rather than re-presenting individual error values in this entry. This change does not affect capability.

Note: If an application has any objects of type **wchar_t**, it must include either **<sys/types.h>** or **<stddef.h>**.

fseek( )

Issue 4: The type of argument *offset* is now defined in full as **long int**. In previous issues it is defined as **long**.

Issue 3 states that ‘‘The *fseek*( ) function does not, by itself, extend the size of of a file’’. This sentence is deleted from Issue 4, although no change of capability is implied.

Issue 4 states that error returns are only generated either when the stream is unbuffered, or if the stream buffer needs to be flushed. This is not stated explicitly in Issue 3.

The **ERRORS** section is revised for consistency with *lseek*( ) and *write*( ).

Issue 4 mandates that the [EIO] error be returned if the calling process is a member of a background process group attempting to write to its controlling terminal, TOSTOP is set, the process is neither ignoring nor blocking SIGTTOU, and the process group of the process is orphaned. Detection of this condition in Issue 3 depends on whether an implementation supports job control.

Issue 4 also explains how *fseek*( ) is used with wide-character input/output; this is marked as a WP extension.

Issue 4, Version 2:

The **ERRORS** section has been updated to include the condition where a physical I/O error has occurred in the description of [EIO] for conformance to X/OPEN UNIX.

fsetpos( )

Issue 4: New function in Issue 4.

The *fsetpos*( ) function sets the file position indicator of the specified stream to *pos*, which is a value obtained from an earlier successful call to *fgetpos*( ). This interface is similar to *fseek*( ), except that the contents of *pos* are unspecified; that is, the value is not necessarily a byte offset.

fstat( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows this header to be included.

In the description, the words ‘‘extended security controls’’ are replaced by ‘‘additional or alternative file access control mechanisms’’. No functional change is implied by this rewording.

Issue 4, Version 2:
For conformance to X/OPEN UNIX, the **ERRORS** section has been updated to include the mandatory error [EIO] when a physical I/O error has occurred, and the optional error [EOVERFLOW] when one of the values is too large to store in the area pointed to by *buf*.

fstatvfs( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *statvfs*( ) and *fstatvfs*( ) functions get file system information. It is unspecified whether all members of the **statvfs** structure have meaningful values on all file systems.

fsync( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

In the **APPLICATION USAGE** section, the words ‘‘require a file to be in a known state’’ are replaced by ‘‘require modifications to a file to be completed before continuing’’. No functional change is implied by this rewording.

ftell( )

Issue 4: The function return value is now defined in full as **long int**. In Issue 3 it is defined simply as **long**.

ftime( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ftime*( ) function sets the **time** and **millitm** members of the **timeb** structure. For portability to implementations conforming to earlier versions of this document, *time*( ) is preferred over this function.

ftok( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ftok*( ) function returns an IPC key based on *path* and *id* that is usable in subsequent calls to *msgget*( ), *semget*( ) and *shmget*( ). For maximum portability, *id* should be a single-byte character.

ftruncate( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *truncate*( ) and *ftruncate*( ) functions are used to truncate a regular file to a specified length.

ftw( )

Issue 4: The type of argument *path* is changed from **char*** to **const char***. The argument list for the *fn*( ) function is also defined.

Issue 4 states that tree traversal continues until the tree is exhausted, an invocation of *fn* returns a non-zero value, or some error other than [EACCES], is detected within *ftw*( ). Issue 3 does not contain the caveat about [EACCES], although no functional change is implied as the significance of this error is dependent on the setting of *flag*.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated to account for symbolic links for X/OPEN UNIX conformance. The **DESCRIPTION** section was further updated to note that *ftw*( ) uses at most one file descriptor for each level in the tree, and constrains *ndirs* to the range from 1 to {OPEN_MAX}.

The **RETURN VALUE** section has been updated to describe the case when *ftw*( ) encounters an error other than [EACCESS].

The **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

fwrite( )

Issue 4: The type of argument *ptr* is changed from **void*** to **const void***.

The description is changed to make it clear that the function advances the file-position indicator by the number of bytes successfully written rather than the number of characters, which could include multi-byte sequences.

gamma( )

Issue 4: This interface is marked TO BE WITHDRAWN, as it is functionally equivalent to *lgamma*( ). Hence the description is also changed to refer to the *lgamma*( ) entry.

gcvt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *ecvt*( ).

getc( )

Issue 4: The description in Issue 4 is updated to make it clear that the function returns a byte value. Issue 3 states that a character is returned, which can be misinterpreted to mean a multi-byte sequence in locales that supports multi-byte codesets.

getchar( )

Issue 4: The argument list is explicitly defined as **void**.

The description in Issue 4 is updated to make it clear that the function returns a byte value. Issue 3 states that a character is returned, which can be misinterpreted to mean a multi-byte sequence in locales that supports multi-byte codesets.

getcontext( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getcontext*( ) and *setcontext*( ) functions get and set the current user context which includes the contents of the calling process' machine registers, the signal mask, and current execution stack.

A portable application cannot assume that the context includes any process-wide static data, possibly including *errno*. Users manipulating contexts should take care to handle these explicitly when required.

getcwd( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Issue 4 states that if *buf* is a null pointer, the behaviour of *getcwd*( ) is undefined. Issue 3 recommends that *getcwd*( ) should not be invoked with a null pointer as this capability may be subject to withdrawal (without explaining what the capability is).

To dig even further back into history, Issue 2 defined that if *buf* is a null pointer, *getcwd*( ) obtains *size* bytes of space using *malloc*( ) and returns a pointer thereto. It is this capability that is not recommended in Issue 3 and undefined in Issue 4.

While many implementations may support the capability defined in Issue 2, portable applications should not rely on it.

getdate( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getdate*( ) function is used to convert a string representation of a date or time into a broken-down time.

Historical versions of *getdate*( ) did not require that **<time.h>** declare the external variable *getdate_err*, but it is now required. X/Open encourages application writers to remove declarations of *getdate_err*, and instead incorporate the declaration by including **<time.h>**.

getdtablesize( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getdtablesize*( ) function is used to get the file descriptor table size, and is equivalent to a call to *getrlimit*( ) with the RLIMIT_NOFILE option.

getegid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getenv( )

Issue 4: The type of argument *name* is changed from **char*** to **const char***.

The description in Issue 4 is changed to indicate that the return string:

- must not be modified by an application
- may be overwritten by subsequent calls to *getenv*( ) or *putenv*( )
- is not overwritten by calls to other X/Open system interfaces.

This does not represent a change in capability, but it does identify extra features of the interface, of which programmers developing portable applications should be aware.

geteuid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getgid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getgrent( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endgrent*( ).

getgrgid( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

A note is added to the **APPLICATION USAGE** section advising that applications wishing to check for errors should set *errno* to zero before calling *getgrgid*( ). If *errno* is set on return, an error occurred.

getgrnam( )

Issue 4: The type of argument *name* is changed from **char*** to **const char***.

The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

A note is added to the **APPLICATION USAGE** section advising that applications wishing to check for errors should set *errno* to zero before calling *getgrnam*( ). If *errno* is set on return, an error occurred.

getgroups( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

A return value of zero is no longer permitted as {NGROUPS_MAX} must be greater than zero. This change in Issue 4 is forced by alignment with the FIPS requirements.

gethostid( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *gethostid*( ) function retrieves a 32-bit identifier for the current host. X/Open does not define the domain in which the return value is unique.

getitimer( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getitimer*( ) and *setitimer*( ) functions get and set the value of an interval timer.

getlogin( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The argument list is explicitly defined as **void**.

The description is updated to state explicitly that the return value is a pointer to a string giving the user name, rather than simply a pointer to the user name as stated in Issue 3.

The **APPLICATION USAGE** section is changed to refer to *getpwuid*( ) rather than *cuserid*( ), which will be withdrawn in a future issue.

getmsg( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getmsg*( ) and *getpmsg*( ) functions retrieve the message at the head of the STREAM head read queue associated with a STREAMS file.

getopt( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section and **<stdio.h>** is deleted.

The type of argument *argv* is changed from **char**** to **char* const []**.

The integer *optopt* is added to the list of external data items.

The description is largely rewritten, without functional change, for alignment with the POSIX-2 standard, although the following differences should be noted:

- If the function detects a missing option argument, it returns a colon (:) and sets **optopt** to the option character.
- The termination conditions under which *getopt*( ) returns **–1** are extended. Also note that the termination condition is explicitly –1, rather than the value of EOF.

getpagesize( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getpagesize*( ) function returns the current page size. It is equivalent to *sysconf(_SC_PAGE_SIZE)* and *sysconf(_SC_PAGESIZE)*.

getpass( )

Issue 4: The interface is marked TO BE WITHDRAWN, because of its misleading name and because it provides dubious capability.

The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *prompt* is changed from **char*** to **const char***.

In the **DESCRIPTION** section, reference to the character special file **/dev/tty** is replaced by the phrase ‘‘the process’ controlling terminal’’. Issue 4 is more correct, although there is no difference in implementation terms.

In Issue 4 the word ‘‘characters’’ is replaced by ‘‘bytes’’, to indicate that this interface deals solely in single-byte values.

getpgid( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getpgid*( ) function returns the process group ID of the process whose process IS is equal to *pid*.

getpgrp( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getpid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getpmsg( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *getmsg*( ).

getppid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getpriority()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getpriority*() and *setpriority*() functions get and set the scheduling priority of a process, process group or user.

getpwent()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endpwent*().

getpwnam()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The type of argument *name* is changed from **char*** to **const char***.

The **APPLICATION USAGE** section is expanded as follows:

- It warns that the return value may point to a static area that is overwritten by subsequent calls to *cuserid*(), *getpwnam*() or *getpwuid*().
- It advises that applications wishing to check for errors should set *errno* to zero before calling *getgrnam*(). If *errno* is set on return, an error occurred.
- It identifies how the various names associated with a process can be determined.

getpwuid()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

A note is added to the **APPLICATION USAGE** section advising that applications wishing to check for errors should set *errno* to zero before calling *getgrnam*(). If *errno* is set on return, an error occurred.

getrlimit()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getrlimit*() and *setrlimit*() functions obtain and set the limits on consumption of a variety of resources for the calling process. Processes are able to set a soft limit to any value less than a hard limit. Processes are able to lower a hard limit, but are not able to raise a hard limit unless they have appropriate privileges.

getrusage( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getrusage*( ) function provides measures of the resources used by the current process or its terminated and waited-for children.

gets( )

Issue 4: The description in Issue 4 is updated to make it clear that the function reads bytes. Issue 3 states that characters are read, which can be misinterpreted to mean multi-byte sequences in locales that support multi-byte codesets.

getsid( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getsid*( ) function gets the process group ID of the process that is the session leader of the process specified by *pid*.

getsubopt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getsubopt*( ) function parses suboption arguments in a flag argument that was initially parsed by *getopt*( ).

gettimeofday( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *gettimeofday*( ) function obtains the current time, expressed as seconds and microseconds since 00:00 Coordinated Universal Time (UTC), January 1, 1970, and stores it in the **timeval** structure pointed to by *tp*.

getuid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

getutxent( )

Issue 4, Version 2:

The *getutxent*( ), *getutxid*( ) and *getutxline*( ) functions were first introduced in Issue 4, Version 2 (X/OPEN UNIX), to get user accounting database entries.

Refer to *endutxent*( ).

getw( )

Issue 4: No changes are made to this interface in Issue 4.

getwc( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *getc*( ), except that it returns a wide-character value. Conversion from multi-byte to wide-character form is done automatically.

getwchar( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *getchar*( ), except that it returns a wide-character value. Conversion from multi-byte to wide-character form is done automatically.

getwd( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *getwd*( ) function determines an absolute pathname of the current working directory of the calling process. For portability to implementations conforming to earlier versions of this document, *getcwd*( ) is preferred over this function.

glob( )

Issue 4: New function in the POSIX.2 C-language Binding feature group in Issue 4.

The *glob*( ) function is a pathname generator that accepts a pattern used for filename expansion and returns a list of pathnames that match. For example, to obtain a list of files that match the pattern:

```
*.r
```

glob( ) can be called as follows:

```
glob("*.r", GLOB_NOCHECK, NULL, &globbuf);
```

The first argument contains the pattern to be matched. The second argument is a flag, where GLOB_NOCHECK indicates that if *pattern* does not match any pathname, then a list containing only the pattern is returned. The third argument is a pointer to an optional error handling function, and the final argument is a pointer to a structure of type **glob_t**.

On return from this call, **globbuf.gl_pathc** contains a count of the paths matched, and **globbuf.gl_pathv** contains a pointer to the matched pathnames. The field **globbuf.gl_offs** can be set on input to reserve slots at the beginning of **gl_pathv**.

gmtime( )

Issue 4: The type of argument *timer* is changed from **time_t*** to **const time_t***.

grantpt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *grantpt*( ) function changes the mode and ownership of the slave pseudo-terminal device associated with the master pseudo-terminal counter part.

hcreate( )

Issue 4: The type of argument *nel* is changed from **unsigned** to **size_t**.

hdestroy( )

Issue 4: The argument list is explicitly defined as **void**.

hsearch( )

Issue 4: The type of argument *nel* in the declaration of the *hcreate*( ) function is changed from **unsigned** to **size_t**, and the argument list is explicitly defined as **void** in the declaration of the *hdestroy*( ) function.

The type of the comparison key is explicitly defined as **char***, the type of *item.data* is explicitly defined as **void***, and a statement is added indicating that *hsearch*( ) uses *strcmp*( ) as the comparison function.

Issue 4 defines [ENOMEM] as an error that may be returned by this function. This is not indicated in Issue 3.

hypot( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if the result overflows or underflows.

iconv( )

Issue 4: New function in Issue 4.

This interface provides code conversion facilities at the application level (see the X/Open Internationalisation Guide, Version 2 for more details).

iconv_close( )

Issue 4: New function in Issue 4.

This interface is the code conversion deallocation function, which is used subsequent to calls to *iconv*( ) (see the X/Open Internationalisation Guide, Version 2 for more details).

iconv_open()

Issue 4: New function in Issue 4.

This interface is the code conversion allocation function, which is used prior to calls to *iconv*( ) (see the X/Open Internationalisation Guide, Version 2 for more details).

ilogb()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ilogb*( ) function returns the unbiased exponent part of *x*. The call *ilogb(x)* is equivalent to (**int**)*logb(x)*.

index()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *index*( ) function is identical to *strchr*( ). For portability to implementations conforming to earlier versions of this document, *strchr*( ) is preferred over this function.

initstate()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *initstate*( ) function is one of the pseudo-random number generator functions, along with *random*( ), *setstate*( ) and *srandom*( ). While some implementations of *random*( ) have written messages to standard error, they do not conform to this document.

insque()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *insque*( ) and *remque*( ) functions manipulate queues built from doubly-linked lists. These queues can be circular or linear.

Historical implementations of these functions described the arguments as being of type **struct qelem *** rather than of type **void ***. This structure was commonly defined in **<search.h>** as simply the two (forward and backward) links, and no room for a data pointer or storage. If this method had been carried forward with an ISO C compiler and the historical function prototype, in order to avoid compilation warnings, most applications would require changes to cast pointers to the structures actually used to become pointers to **struct qelem**. Specifying **void *** as the argument type will not require applications to change (unless they specifically reference **struct qelem** and depend upon its definition in **<search.h>**).

ioctl( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ioctl*( ) function performs a variety of control functions on STREAMS devices. On non-STREAMS devices, the functions performed are implementation-defined.

isalnum( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isalpha( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isascii( )

Issue 4: No changes are made to this interface in Issue 4.

isastream( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *isastream*( ) function tests whether *fildes*, an open file descriptor, is associated with a STREAMS-based file.

isatty( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

iscntrl( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isdigit( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isgraph( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

islower( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isnan( )

Issue 4: No changes are made to this interface in Issue 4.

isprint( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

ispunct( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isspace( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

isupper( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

iswalnum( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isalnum*( ), except that it accepts a wide-character code as input.

iswalpha( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isalpha*( ), except that it accepts a wide-character code as input.

iswcntrl( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *iscntrl*( ), except that it accepts a wide-character code as input.

iswctype( )

Issue 4: New function in Issue 4 (WPI)

This interface is used with *wctype*( ) to provide classification facilities for all character classes in a locale, including those for which there is no direct *isw**( ) function. The interface accepts a **charclass** argument of type **wctype_t**, which is a value returned by a previous successful call to *wctype*( ).

For example, assume a program wishes to check for diacritical marks in locales that support such objects:

```
...
int      count = 0;
wint_t   wchar;
wctype_t charclass = wctype("diacritical");
...
if (charclass != (wctype_t)0))
    while ((wchar = getwchar()) != WEOF)
        if (iswctype(wchar, charclass))
        count += 1;

printf ("%d diacritical marks found\n", count);

...
```

In the above code sequence, the call to *wctype*( ) only returns a valid value when the program is running in a locale that supports the identified character class; that is, for which the class **diacritical** is defined in the *localedef* source file from which the locale is generated.

iswdigit( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isdigit*( ), except that it accepts a wide-character code as input.

iswgraph( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isgraph*( ), except that it accepts a wide-character code as input.

iswlower( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *islower*( ), except that it accepts a wide-character code as input.

iswprint( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isprint*( ), except that it accepts a wide-character code as input.

iswpunct( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *ispunct*( ), except that it accepts a wide-character code as input.

iswspace( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isspace*( ), except that it accepts a wide-character code as input.

iswupper( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isupper*( ), except that it accepts a wide-character code as input.

iswxdigit( )

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *isxdigit*( ), except that it accepts a wide-character code as input.

isxdigit( )

Issue 4: The text of the **DESCRIPTION** and **RETURN VALUE** sections is revised in Issue 4, although there are no functional differences between this issue and Issue 3. Operation in the POSIX locale is now described elsewhere in the XSH specification.

j0( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if underflow occurs.

jrand48()

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *xsubi* is defined in Issue 4 as an array of **unsigned short int**. In Issue 3 it is simply (but equally) defined as an array of **unsigned short**. Similarly, the type of *jrand48*( ) is changed from **long** to **long int**.

kill()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The description is changed to indicate that {POSIX_SAVED_IDS} is defined on all XSI-conformant systems and, because of this, that the saved set-user-ID of the calling process is checked in place of its effective user ID. Issue 3 does not mandate that {POSIX_SAVED_IDS} must be set.

killpg()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *killpg*( ) function sends the specified signal to the specified process group.

l64a()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *a64l*( ).

labs()

Issue 4: New function in Issue 4.

The *labs*( ) function computes the absolute value of a long integer.

lchown()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *lchown*( ) function has the same effect as *chown*( ) except in the case where the named file is a symbolic link. In this case, *lchown*( ) changes the ownership of the symbolic link file itself, while *chown*( ) changes the ownership of the file or directory to which the symbolic link refers.

lcong48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *param* is defined in Issue 4 as an array of **unsigned short int**. In Issue 3 it is simply (but equally) defined as an array of **unsigned short**.

ldexp( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] is set if the value overflows. Issue 3 defines that an implementation might set [ERANGE] in these circumstances.

ldiv( )

Issue 4: New function in Issue 4.

The *ldiv*( ) function is equivalent to *div*( ), except that it computes the quotient and remainder of a long division.

lfind( )

Issue 4: The type of the function return value is changed from **char*** to **void***, the type of the *key* and *base* arguments is changed from **void*** to **const void***, and argument declarations for the *compar*( ) function are added.

lgamma( )

Issue 4: This page no longer points to the *gamma*( ) entry, but contains all information relating to the *lgamma*( ) function on this page.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be set if the result underflows. The handling of underflow conditions is not specified in Issue 3.

link( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of arguments *path1* and *path2* are changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, the [EXDEV] error may also be returned if *path1* refers to a STREAM, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a

symbolic link.

loc1

Issue 4: The header **<regexp.h>** is added to the **SYNOPSIS** section.

localeconv( )

Issue 4: New function in Issue 4.

This function sets the components of an object of type **struct lconv** with values appropriate to the formatting of numeric quantities (monetary and otherwise) in the current locale. This is a lower-level interface than *strfmon*( ) (see the X/Open Internationalisation Guide, Version 2 for further details).

localtime( )

Issue 4: The type of argument *timer* is changed from **time_t*** to **const time_t***.

lockf( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *lockf*( ) function allows sections of a file to be locked with advisory-mode locks. Record-locking should not be used in conjunction with *fopen*( ), *fread*( ), *fwrite*( ) and other *stdio* functions, as buffering may cause unexpected results. Applications should use the more primitive, non-buffered functions, such as *open*( ).

locs

Issue 4: The header **<regexp.h>** is added to the **SYNOPSIS** section. This variable has been marked **TO BE WITHDRAWN**.

log( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] is set if *x* is negative. Issue 3 indicates that an implementation might set this error.
- [EDOM] may be set if the value of *x* is NaN. Issue 3 indicates that [EDOM] might be set if *x* is negative or zero.
- [ERANGE] may be set if the value of *x* is zero. Issue 3 indicates that [ERANGE] might be set if *x* is zero, the logarithm of *x* could not be represented or the result would cause overflow. These last two conditions are not described in Issue 4.

log10( )

Issue 4: The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] is set if *x* is negative. This condition is not described in Issue 3.
- [EDOM] may be set if the value of *x* is NaN. Issue 3 indicates that [EDOM] might be set if *x* is not greater than zero.
- [ERANGE] may be set if the value of *x* is zero. Issue 3 indicates that [ERANGE] might be set if *x* is zero, the logarithm of *x* could not be represented or the result would cause overflow. These last two conditions are not described in Issue 4.

log1p( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *log1p*( ) function computes the natural logarithm of 1.0 + *x*. The value of *x* must be greater than –1.0.

logb( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *logb*( ) function determines the radix-independent exponent of *x*.

_longjmp( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *_longjmp*( ) and *_setjmp*( ) functions provide non-local goto capability and are identical to *longjmp*( ) and *setjmp*( ) respectively, with the additional restriction that *_longjmp*( ) and *_setjmp*( ) do not manipulate the signal mask.

The *_longjmp*( ) and *_setjmp*( ) functions are included to support programs written to historical system interfaces. New applications should use *siglongjmp*( ) and *sigsetjmp*( ) respectively.

longjmp( )

Issue 4: No functional changes are made to this interface in Issue 4, although note that implications of volatile-qualified types are described.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance and discusses the valid possibilities for the resulting state of the signal mask. Since the effect of *longjmp*( ) and *setjmp*( ) on the signal mask is unspecified, applications that depend upon the value of the signal mask should investigate the use of *_longjmp*( ) and *_setjmp*( ) (which never modify the mask), or *siglongjmp*( ) and *sigsetjmp*( ) (which can save and restore the mask under application control). For portability to implementations conforming to earlier versions of this document, *siglongjmp*( ) and *sigsetjmp*( ) are preferred.

lrand48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The argument list is explicitly defined as **void**.

lsearch( )

Issue 4: The type of argument *key* in the declaration of the *lsearch*( ) function is changed from **void*** to **const void***, the type arguments *key* and *base* have been changed from **void*** to **const void*** in the declaration of the *lfind*( ) function, and the arguments to *compar*( ) are defined for both functions.

lseek( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

lstat( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *lstat*( ) function behaves the same as *stat*( ), except when *path* refers to a symbolic link. In that case *lstat*( ) returns information about the link, while *stat*( ) returns information about the file the link references.

makecontext( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *makecontext*( ) and *swapcontext*( ) functions manipulate user contexts initialised using *getcontext*( ).

malloc( )

Issue 4: Issue 4 states that if *size* is zero, either a null pointer or a unique pointer that can be successfully passed to *free*( ) is returned. This is not defined in Issue 3.

There is now no requirement for the implementation to support the inclusion of **<malloc.h>**.

mblen( )

Issue 4: New function in Issue 4.

The *mblen*( ) function determines the number of bytes in a (possibly multi-byte) character, according to information in the LC_CTYPE category of the current locale.

mbstowcs( )

Issue 4: New function in Issue 4.

The *mbstowcs*( ) function converts a character string, possibly containing multi-byte characters, to a wide-character string, according to information in the LC_CTYPE category of the current locale.

mbtowc( )

Issue 4: New function in Issue 4.

This function is similar to *mbstowcs*( ), except that it only converts a single, possibly multi-byte character to a wide-character code.

memccpy( )

Issue 4: The type of argument *s2* is changed from **void*** to **const void***.

Issue 3 states, as an application note, that compliant implementations also support inclusion of the header **<memory.h>**. This is done for backwards compatibility with XPG2. Issue 4 does not promise that this header is provided, which may cause applications dependent on this header to fail when compiled on an Issue 4 system.

This note applies to all the memory functions.

memchr( )

Issue 4: The type of argument *s* is changed from **void*** to **const void***.

memcmp( )

Issue 4: The type of arguments *s1* and *s2* are changed from **void*** to **const void***.

memcpy( )

Issue 4: The type of argument *s2* is changed from **void*** to **const void***.

A note is added to Issue 4 indicating that *memcpy*( ) does not check for overflow in the receiving memory area. This is also true in Issue 3, although the warning is not stated explicitly.

memmove( )

Issue 4: New function in Issue 4.

The *memmove*( ) function allows bytes to be copied between overlapping areas. Otherwise, it is exactly like the *memcpy*( ) function.

memset( )

Issue 4: No change is made to this interface in Issue 4.

mkdir()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

mkfifo()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The type of argument *path* is changed from **char*** to **const char***.

The description of [EACCES] is updated to indicate that this error is also returned if write permission is denied to the parent directory.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

mknod()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *mknod*( ) function creates a new directory, or a new special or regular file named by the pathname argument. The only portable use of *mknod*( ) is to create a FIFO-special file. If *mode* is not S_IFIFO or *dev* is not 0, the behaviour of *mknod*( ) is unspecified.

For portability to implementations conforming to earlier versions of this document, *mkfifo*( ) is preferred over this function for making FIFO special files.

mkstemp( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *mkstemp*( ) function replaces the contents of the string pointed to by *template* by a unique filename, and returns a file descriptor for the file open for reading and writing. This function thereby prevents any race conditions between testing whether the file exists and opening it.

For portability to implementations conforming to earlier versions of this document, *tmpfile*( ) is preferred over this function.

mktemp( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *mktemp*( ) function replaces the contents of the string pointed to by *template* by a unique filename. This function does not open a file of the name returned, therefore it is possible for another process to create a file of the same name. Using *mkstemp*( ) avoids this problem.

For portability to implementations conforming to earlier versions of this document, *tmpnam*( ) is preferred over this function.

mktime( )

Issue 4: The description in Issue 4 is changed to indicate the possible settings of *tm_isdst*, and reference to setting of *tm_sec* for leap seconds or double leap seconds is removed (although this capability is still supported).

mmap( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *mmap*( ) function establishes a mapping between a process' address space and a file.

modf( )

Issue 4: The name of the first argument is changed from *value* to *x* in the **SYNOPSIS** section.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may be returned if the result underflows.

mprotect( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *mprotect*( ) function changes the access protections on the memory mappings specified by the range *[addr, addr + len)*, rounding up *len* to the next multiple of the page size as returned by *sysconf*( ).

mrand48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The argument list is explicitly defined as **void**.

msgctl( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

msgget( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

msgrcv( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The type of argument *msgtype* is now defined in full as **long int**.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

msgsnd( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The type of argument *msgp* is changed from **void*** to **const void***.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

msync( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *msync*( ) function writes all modified copies of pages over the range *[addr, addr + len)* to the underlying hardware, or invalidates any copies so that further references to the pages will be obtained by the system from their permanent storage locations. It should be used by programs requiring that a memory object be in a known state.

Normal system activity can cause pages to be written to disk, therefore there are no guarantees that *msync*( ) is the only control over when pages are or are not written.

munmap( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *munmap*( ) function removes the mappings for pages in the range *[addr, addr + len)*, rounding the *len* argument up to the next multiple of the page size as returned by *sysconf*( ).

nextafter( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *nextafter*( ) function computes the next representable double-precision floating-point value following *x* in the direction of *y*.

nftw()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *nftw*() function recursively descends the directory hierarchy rooted at *path*. The *nftw*() function has a similar effect to *ftw*() except that it takes an additional flags argument that provides additional control over the hierarchy traversal.

nice()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

A statement is added to Issue 4 indicating that the *nice* value can only be lowered by a process with appropriate privilege. This is implied in Issue 3, by virtue of the fact [EPERM] is defined to be returned if *incr* is negative, but it is not stated explicitly.

Issue 4, Version 2:

The **RETURN VALUE** section has been updated for X/OPEN UNIX conformance to indicate that a process' *nice* remains unchanged if an error is detected.

nl_langinfo()

Issue 4: The header **<nl_types.h>** is removed from the **SYNOPSIS** section. Existing applications that include this header should continue to compile on an Issue 4 system without needing to be modified.

nrand48()

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of *xsubi* is defined in Issue 4 as **unsigned short int**. This is a pedantic change and should not affect existing applications.

open()

Issue 4: The headers **<sys/types.h>** and **<sys/stat.h>** are no longer required explicitly; these headers are optional on XSI-conformant systems, although the POSIX-1 standard shows them to be included.

The type of argument *path* is changed from **char*** to **const char***.

Various wording changes are made to the description in Issue 4 to improve clarity and to align the text with the POSIX-1 standard. No functional changes are implied by this rewording.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

O_NDELAY is removed from the list of *oflag* values. As this flag is marked WITHDRAWN in Issue 3, it should not affect existing applications (unless they are XPG2-compliant, or older).

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to define the use of open flags with STREAMS files, and to identify special considerations when opening the master side of a pseudo-terminal.

The **ERRORS** section has been updated for X/OPEN UNIX conformance to add [EIO], [ELOOP] and [ENOSR] as mandatory errors, and [EAGAIN], [ENAMETOOLONG] and [ENOMEM] as optional errors.

opendir()

Issue 4:

The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The type of argument *dirname* is changed from **char*** to **const char***.

Issue 3 contained the following statement:

> ''The type **DIR**, which is defined in **<dirent.h>**, represents a *directory stream*, which is an ordered sequence of all directory entries in a particular directory.''

This is moved to the XBD specification in Issue 4. No change of capability is implied.

The generation of an [ENOENT] error when *dirname* points to an empty string is made mandatory. Detection of this condition is defined as optional in Issue 3.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

openlog()

Issue 4, Version 2:

This function was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *closelog*( ).

optarg

Issue 4:

Entry derived from *getopt*( ) in Issue 3, with the following changes:

- Item *optopt* is added to the list of external data items.

pathconf()

Issue 4: The type of argument *path* is changed from **char*** to **const char***. Also the return value of both functions is changed from **long** to **long int**.

Issue 4 states (in notes 2, 4 and 6) that it is unspecified whether an implementation supports an association of the variable name with the specified file. Issue 3 states that the behaviour of the interface is undefined in these circumstances.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

pause()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The argument list is explicitly defined as **void**.

Issue 4 states that since the *pause*() function suspends indefinitely unless interrupted by a signal, there is no successful completion return value. Issue 3 did not contain the words ''unless interrupted by a signal'', although this is stated elsewhere in the text. No change of capability is implied.

pclose()

Issue 4: As *pclose*() is specified in the POSIX-2 standard, it is no longer marked as an extension in the XSH specification.

The simple description given in Issue 3 is replaced with a more complete description in this issue. In particular, interactions between this function and *wait*() and *waitpid*() are defined. The essential capability of the interface is the same, but application developers are advised to check the Issue 4 entry as it spells out hitherto unstated features of the interface.

perror()

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

A paragraph is added to Issue 4 defining the effects of this function on the *st_ctime* and *st_mtime* fields of the standard error stream. This is implicit capability that is simply not stated in Issue 3.

The language for error message strings is given as implementation-defined in Issue 3. In Issue 4, message strings are defined as language-dependent; that is, they are dependent on the locale in which the program that calls *perror*() is running.

pipe()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate that certain dispositions of *fildes*[0] and *fildes*[1] are unspecified.

poll()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *poll*( ) function provides applications with a mechanism for multiplexing input/output over a set of file descriptors. The *poll*( ) function supports regular files, terminal and pseudo-terminal devices, STREAMS-based files, FIFOs and pipes. (It also supports sockets, if the XPG4 Sockets component is supported.)

popen()

Issue 4: As *popen*( ) is specified in the POSIX-2 standard, it is no longer marked as an extension in the XSH specification.

The type of arguments *command* and *mode* are changed from **char*** to **const char***.

The description of this interface in Issue 4 is completely rewritten for alignment with the POSIX-2 standard, although it describes essentially the same capability as Issue 3.

The **ERRORS** section is added; it includes an indication that this interface may return any of the errors defined for the *fork*( ) or *pipe*( ) functions.

The **APPLICATION USAGE** section is extended, with only notes about buffer flushing being retained from Issue 3. Application developers are advised to read this section as it contains useful background information about implementation of the interface.

pow()

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- Issue 4 states that [EDOM] is set if the value of *x* is negative and *y* is non-integral. Issue 3 defines that the *pow*( ) function might fail under these circumstances.
- Issue 4 states that [ERANGE] is set if the value to be returned would cause overflow. Again, Issue 3 defines that the *pow*( ) function might fail and set [ERANGE].

printf( )

Issue 4: Refer to *fprintf*( ).

ptsname( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ptsname*( ) function returns the name of the slave pseudo-terminal device associated with a master pseudo-terminal device.

The value returned may point to a static data area that is overwritten by each call to *ptsname*( ).

putc( )

Issue 4: No changes are made to this interface in Issue 4.

putchar( )

Issue 4: No changes are made to this interface in Issue 4.

putenv( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *string* is changed from **char*** to **const char***.

putmsg( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *putmsg*( ) and *putpmsg*( ) functions create a message from a process buffer(s) and send the message to a STREAMS file.

puts( )

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

The description in Issue 4 is updated to make it clear that this function manipulates bytes. Issue 3 incorrectly referred to characters, which can be misinterpreted to mean multi-byte sequences in locales that support multi-byte codesets.

pututxline( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endutxent*( ).

putw()

Issue 4: No changes are made to this interface in Issue 4.

putwc()

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *putc*( ), except that it accepts a wide character as input (encoded as a value of type **wint_t**), which it converts to a possibly multi-byte sequence and outputs to *stream*.

Note that the only difference between *putwc* and *fputwc* is that the former can be implemented as macro, meaning that *stream* may be evaluated more than once. Thus, arguments should never be expressions with side-effects, for example:

```
putwc(wc, *f++);
```

For this reason, application developers are recommended to use the *fputwc*( ) function in preference to *putwc*( ).

putwchar()

Issue 4: New function in Issue 4 (WPI).

This interface is functionally equivalent to *putchar*( ), except that it accepts a wide character as input (encoded as a value of type **wint_t**), which it converts to a possibly multi-byte sequence and outputs to the standard output stream.

qsort()

Issue 4: The arguments to function *compar*( ) are formally defined in the **SYNOPSIS** section. No functional change is implied by this.

raise()

Issue 4: New function in Issue 4.

The *raise*( ) function allows a program to send a signal, identified by the argument *sig*, to itself. For example:

```
raise(SIGUSR1);
```

rand()

Issue 4: The argument list of the *rand*( ) function is explicitly defined as **void**.

The definition of *srand*( ) is added to the **SYNOPSIS** section, and the argument *seed* is explicitly defined as **unsigned int**.

random()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *initstate*( ).

read( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *buf* is changed from **char*** to **void***, and the type of argument *nbyte* is changed from **unsigned** to **size_t**.

Issue 4 states that the result is implementation-dependent if *nbyte* is greater than {SSIZE_MAX}. This limit is defined by the constant {INT_MAX} in Issue 3. This is unlikely to have any practical implications for applications, as the minimum acceptable value for both constants is defined to be 32767.

The last paragraph of the description in Issue 4 states that if *read*( ) is interrupted by a signal after it has successfully read some data, it returns the number of bytes read. In Issue 3 it is optional whether *read*( ) returns the number of bytes read, or whether it returns –1 with *errno* set to [EINTR].

Issue 4 mandates that the [EIO] error is returned if the calling process is a member of a background process group attempting to read from its controlling terminal, the process is ignoring or blocking SIGTTIN or the process group is orphaned. Detection of this condition in Issue 3 depends on whether an implementation supports job control.

Issue 4, Version 2:

The *readv*( ) function has been added to the **SYNOPSIS** section, for X/OPEN UNIX conformance, and an operational description for *readv*( ) has been added to the **DESCRIPTION** section, along with appropriate changes to the **RETURN VALUE** and **ERRORS** sections.

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to describe reading data from STREAMS files.

The **ERRORS** section was reorganised to describe errors that apply generally to both *read*( ) and *readv*( ), and errors that apply specifically to *readv*( ). The [EBADMSG], [EINVAL] and [EISDIR] errors are also added.

readdir( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The description is updated to indicate that after a call to *fork*( ) either the parent or the child (but not both) may continue processing the directory stream; otherwise the results are undefined. This warning is not in Issue 3.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate the disposition of certain fields in **struct direct** when an entry refers to a symbolic link.

The [ENOENT] error has been added to the **ERRORS** section.

readlink( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *readlink*( ) function reads the contents of a symbolic link into *buf*.

Portable applications should not assume that the returned contents of the symbolic link are null-terminated.

readv( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *read*( ).

realloc( )

Issue 4: The description is updated to align with the ISO C standard. These changes are largely superficial, although they should be noted as they may have a marginal effect on some applications. Specifically:

- The order and contiguity of storage allocated by successive calls to this function is unspecified.
- Each allocation yields a pointer to an object disjoint from any other object.
- The returned pointer points to the lowest byte address of the allocation.
- If *size* is zero, either a null pointer or a unique pointer that can be successfully passed to *free*( ) is returned.

realpath( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *realpath*( ) function determines an absolute pathname that names the same file as *file_name*, whose resolution does not involve ".", ".." or symbolic links.

re_comp( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *re_comp*( ) and *re_exec*( ) functions compile and execute regular expressions.

These interfaces are marked **TO BE WITHDRAWN**, because improved capability is now provided by interfaces introduced for alignment with the POSIX-2 standard (see *regcomp*( )). For portability to implementations conforming to earlier versions of this document, *regcomp*( ) and *regexec*( ) are preferred over these functions.

regcmp( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *regcmp*( ) and *regex*( ) functions compile and execute regular expressions.

These interfaces are marked **TO BE WITHDRAWN**, because improved capability is now provided by interfaces introduced for alignment with the POSIX-2 standard (see *regcomp*( )). For portability to implementations conforming to earlier versions of this document, *regcomp*( ) and *regexec*( ) are preferred over these functions.

regcomp( )

Issue 4: New function in the POSIX.2 C-language Binding feature group in Issue 4.

Two versions of regular expressions are supported in Issue 4:

- The historical simple regular expressions, which provide backward compatibility with earlier issues of the XSH specification.
- The improved internationalised version that complies with the POSIX-2 standard.

The first type is supported by the *regexp*( ) functions, but note that these will be withdrawn in a future issue of the specification.

The second type is supported by the *regcomp*( ) functions, which include:

- *regcomp*( ), which compiles a regular expression from a pattern string and stores the results in a structure of type **regex_t**
- *regexec*( ), which compares a string against a compiled regular expression produced by *regcomp*( )
- *regfree*( ), which frees any memory allocated by *regcomp*( )
- *regerror*( ), which provides a mapping from error codes produced by the other regular expression functions to printable strings.

Examples of how these functions can be used are given in the XSH specification.

regex( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *regcmp*( ).

regexp( )

Issue 4: The interface is marked TO BE WITHDRAWN, because improved capability is now provided by interfaces introduced for alignment with the POSIX-2 standard (see *regcomp*( )).

The type of arguments *instring*, *endbuf*, *string*( ) and *expbuf*( ) are changed from **char*** to **const char***.

Issue 4 states that the *regexp*( ) functions can be implemented as macros. This has always been true, even though it is not stated explicitly in earlier issues of the XSH specification.

remainder( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *remainder*( ) function returns the floating point remainder $r = x - ny$ when y is non-zero. The behaviour of *remainder*( ) is independent of the rounding mode.

remove( )

Issue 4: The type of argument *path* is changed from **char*** to **const char***.

remque( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *insque*( ).

rename( )

Issue 4: The type of arguments *old* and *new* are changed from **char*** to **const char***.

Issue 4 states that if an error occurs, neither file is changed or created. This is not stated explicitly in Issue 3.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4 defines that [EMLINK] is returned if the file named by *old* is a directory, and the link count of the parent directory of *new* would exceed [LINK_MAX]. This is an omission in Issue 3.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate the results of naming a symbolic link in either *old* or *new*.

The **ERRORS** section has been updated for X/OPEN UNIX conformance to add [EIO] to indicate a physical I/O error has occurred, [ELOOP] to indicate too many symbolic links were encountered during pathname resolution, and [EPERM] or [EACCESS] to indicate a permission check failure when operating on directories with S_ISVTX set. The **ERRORS** section was also updated to add a second [ENAMETOOLONG] condition that may be reported for excessive length of an intermediate result of pathname resolution of a symbolic link.

rewind( )

Issue 4: No changes are made to this interface in Issue 4.

rewinddir()

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The description is updated to indicate that after a call to *fork*( ) either the parent or the child (but not both) may continue processing the directory stream; otherwise the results are undefined. This warning is not in Issue 3.

rindex()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *rindex*( ) function is identical to *strrchr*( ). For portability to implementations conforming to earlier versions of this document, *strrchr*( ) is preferred over this function.

rint()

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *rint*( ) function returns the integral part nearest *x* in the direction of the current rounding mode. The current rounding mode is implementation-dependent.

rmdir()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *path* is changed from **char*** to **const char***.

The description in Issue 4 is extended to indicate that, if the directory is a root directory or a current working directory, it is unspecified whether the function succeeds, or whether it fails and sets *errno* to [EBUSY]. In Issue 3, the behaviour under these circumstances is simply defined as ''implementation-dependent''.

Issue 4 states that if –1 is returned, the directory is not changed.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate the results of naming a symbolic link in *path*.

The **ERRORS** section has been updated for X/OPEN UNIX conformance to add [EIO] to indicate a physical I/O error has occurred, [ELOOP] to indicate too many symbolic links were encountered during pathname resolution, and [EPERM] or [EACCESS] to indicate a permission check failure when operating on directories with S_ISVTX set. The **ERRORS** section was also updated to add a second [ENAMETOOLONG] condition that may be reported for excessive length of an intermediate result of pathname resolution of a symbolic link.

sbrk( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *brk*( ).

scalb( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *scalb*( ) function computes $x * r^n$, where r is the radix of the machine's floating point arithmetic. An application wishing to check for error situations should set *errno* to 0 before calling *scalb*( ). If *errno* is non-zero on return, or the return value is NaN, an error has occurred.

scanf( )

Issue 4: Refer to *fscanf*( ).

seed48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Also in the **SYNOPSIS** section, **unsigned short** is replaced by **unsigned short int**.

seekdir( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems although the POSIX-1 standard shows it to be included.

The type of argument *loc* is expanded to **long int**.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate that a call to *readdir*( ) may produce unspecified results if either *loc* was not obtained by a previous call to *telldir*( ), or if there is an intervening call to *rewinddir*( ).

select( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *select*( ) function indicates which of the specified file descriptors is ready for reading, ready for writing, or has an error condition pending. If the specified condition is false for all of the specified file descriptors, *select*( ) blocks for up to the specified timeout interval until the specified condition is true for at least one of the specified file descriptors. The use of a timeout does not affect any pending timers set up by *alarm*( ), *ualarm*( ) or *settimer*( ).

semctl( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The last argument is now defined by an ellipsis symbol. In previous issues it is defined as a union of the various types required by settings of *cmd*. These are now defined individually in each description of permitted *cmd* settings.

Issue 4, Version 2:

The fourth argument to *semctl*( ) has been moved from the **APPLICATION USAGE** section to the **DESCRIPTION** section, and references to its elements have been made more precise.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

semget( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

semop( )

Issue 4: As interprocess communication is mandatory in Issue 4, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The [ENOSYS] error is removed from the **ERRORS** section.

The type of *nsops* is expanded to **unsigned int**.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The description in Issue 4 is updated to indicate that an implementation does not modify the elements of *sops* unless the application uses implementation-dependent extensions.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

setbuf( )

Issue 4: No changes are made to this interface in Issue 4.

setcontext( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *getcontext*( ).

setgid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

setgrent( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endgrent*( ).

setitimer( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *getitimer*( ).

_setjmp( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *_longjmp*( ).

setjmp( )

Issue 4: This issue states that *setjmp*( ) is a macro or a function; previous issues state that it is a macro. Warnings have also been added about the suppression of a *setjmp*( ) macro definition.

Text describing the accessibility of objects after a *longjmp*( ) call is added to the description in Issue 4. This text is imported from the entry for *longjmp*( ). No change of capability is implied.

Text describing the contexts in which calls to *setjmp*( ) are valid is moved to the **DESCRIPTION** section in Issue 4 from the **APPLICATION USAGE** section in Issue 3.

setkey( )

Issue 4: In Issue 4 this function is part of the Encryption feature group.

The type of argument *key* is changed from **char*** to **const char***.

setlocale( )

Issue 4: The type of argument *locale* is changed from **char*** to **const char***.

LC_MESSAGES is added to the list of *category* values. This category affects what strings are expected or given by commands and utilities for affirmative/negative responses, and the contents of messages (see *catopen*( )).

The name POSIX is added to the list of standard *locale* names. The description of "" is clarified: for XSI-conformant systems, this corresponds to the associated environment variables, *LC_** and *LANG*.

setlogmask( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *closelog*( ).

setpgid( )

Issue 4: As job control is supported on all Issue 4-compliant systems, this interface is no longer marked as OPTIONAL FUNCTIONALITY. The description in Issue 4 is changed to reflect this, and the [ENOSYS] error is removed from the **ERRORS** section.

The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

setpgrp( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *setpgrp*( ) function sets the process group ID of the calling process to the process group ID of the calling process, if the calling process is not already a session leader.

setpriority( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *getpriority*( ).

setpwent( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endpwent*( ).

setregid( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *setregid*( ) function is used to set the real and effective group IDs of the calling process. Only a process with appropriate privileges can set the real group ID and effective group ID to any valid value.

setreuid( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *setreuid*( ) function is used to set the real and effective user IDs of the calling process. A process with appropriate privileges can set either ID to any value. An unprivileged process can only set the effective user ID if the *euid* argument is equal to either the real, effective or saved user ID of the process.

setrlimit( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *getrlimit*( ).

setsid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

The argument list is explicitly defined as **void**.

setstate( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *initstat*( ).

setuid( )

Issue 4: The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

No functional changes are made in Issue 4, but it is worth noting that all references to the saved set-user-ID are marked as extensions. This is because Issue 4 defines this mechanism as mandatory, whereas the POSIX-1 standard defines that it is only

supported if {POSIX_SAVED_IDS} is set.

setutxent( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *endutxent*( ).

setvbuf( )

Issue 4: No changes are made to this interface in Issue 4, but note that because the *setvbuf*( ) function is defined in the ISO C standard, it is no longer marked as an extension.

shmat( )

Issue 4: In Issue 4, this function is part of the Shared Memory feature group.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The type of argument *shmaddr* is changed from **char*** to **const void***.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

shmctl( )

Issue 4: In Issue 4, this function is part of the Shared Memory feature group.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

Issue 4, Version 2:

The **ERRORS** section has been updated for X/OPEN UNIX conformance to include [EOVERFLOW] as an optional error.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

shmdt( )

Issue 4: In Issue 4, this function is part of the Shared Memory feature group.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

The type of argument *shmaddr* is changed from **char*** to **const void***.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

shmget( )

Issue 4: In Issue 4, this function is part of the Shared Memory feature group.

Inclusion of the **<sys/types.h>** and **<sys/ipc.h>** headers is removed from the **SYNOPSIS** section. However, because the POSIX-2 standard defines that headers can be included more than once and in any order, existing applications should continue to compile without needing to be modified.

Note: The IEEE P1003.4 Standards Committee has developed alternative interfaces for interprocess communication. Refer to Section 5.6 on page 102.

sigaction( )

Issue 4: The type of argument *act* is changed from **struct sigaction*** to **const struct sigaction***.

Issue 4 states that the consequence of attempting to set SIG_DFL for a signal that cannot be caught or ignored is unspecified. The [EINVAL] error, describing one possible reaction to this condition, has also been added to the **ERRORS** section.

The *raise* and *signal*( ) functions are added to the list of functions that are either re-entrant or not interruptible by signals. The *fpathconf*( ) function is added to this list and marked as an extension. The *ustat*( ) function is removed from the list, as this function is withdrawn from Issue 4. It is no longer specified whether *abort*( ), *chroot*( ), *exit*( ) and *longjmp*( ) also fall into this category of functions.

Issue 4, Version 2:

The following has been updated for X/OPEN UNIX conformance:

- The **DESCRIPTION** section describes **sa_sigaction**, the member of the **sigaction** structure that is the signal-catching function.
- The **DESCRIPTION** section describes the SA_ONSTACK, SA_RESETHAND, SA_RESTART, SA_SIGINFO, SA_NOCLDWAIT and SA_NODEFER settings of *sa_flags*, and their implications and uses.
- The **DESCRIPTION** section specifies the effect if the action for the SIGCHLD signal is set to SIG_IGN.
- The **DESCRIPTION** section includes additional text describing the effect if the action is a pointer to a function. Additional text covers the case where SA_SIGINFO is set. SIGBUS is added as a signal for which the behaviour of a process is undefined following a normal return from the signal-catching function.
- The **APPLICATION USAGE** section includes additional material describing the use of an alternate signal stack; resumption of the process receiving the signal; coding for compatibility with IEEE Std 1003.1b-1993 (POSIX.1b); and implementations of signal-handling functions in BSD.

sigaddset( )

Issue 4: The **ERRORS** section is changed to indicate that [EINVAL] may be produced by an implementation. In Issue 3 this error is defined as mandatory.

sigaltstack( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *sigaltstack*( ) function allows a process to define and examine the state of an alternate stack of signal handlers. Signals that have been explicitly declared to execute on the alternate stack will be delivered on the alternate stack.

sigdelset( )

Issue 4: The **ERRORS** section is changed to indicate that [EINVAL] may be produced by an implementation. In Issue 3 this error is defined as mandatory.

sigemptyset( )

Issue 4: No changes are made to this interface in Issue 4, although note that spurious text at the end of the **RETURN VALUE** section is removed.

sigfillset( )

Issue 4: No changes are made to this interface in Issue 4, although note that spurious text at the end of the **RETURN VALUE** section is removed.

sighold( )

Issue 4, Version 2:

The *sighold*( ) and *sigignore*( ) functions to add a signal to the signal mask, or set a signal disposition to be ignored, were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *signal*( ).

siginterrupt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *siginterrupt*( ) function is used to change the restart behaviour when a function is interrupted by the specified signal.

The *siginterrupt*( ) function supports programs written to historical system interfaces. A portable application, when being written or rewritten, should use *sigaction*( ) with the SA_RESTART flag rather than *siginterrupt*( ).

sigismember( )

Issue 4: The type of argument *set* is changed from **sigset_t*** to **const sigset_t***.

The **ERRORS** section is changed to indicate that [EINVAL] may be produced by an implementation. In previous issues this error is defined as mandatory.

siglongjmp( )

Issue 4: No changes are made to this interface in Issue 4, although note that implications of volatile-qualified types are added to the description.

signal( )

Issue 4: Because *signal*( ) is specified in the ISO C standard, this function is no longer marked as an extension.

The argument **int** is added to the definition of the *func*( ) function in the **SYNOPSIS** section.

In Issue 3, this interface cross-referred to *sigaction*( ). Issue 4 provides a complete description of the function as defined in the ISO C standard.

Issue 4, Version 2:

The following has been updated for X/OPEN UNIX conformance:

- The *sighold*( ), *sigignore*( ), *sigpause*( ), *sigrelse*( ) and *sigset*( ) functions have been added to the **SYNOPSIS** section, and the **DESCRIPTION** section has been updated appropriately to describe those interfaces.
- The **RETURN VALUE** section has been updated to describe the possible returns from the *sigset*( ) function specifically, and all of the above functions generally.
- The **ERRORS** section has been reorganised to describe possible error returns from each of the functions individually.
- The **APPLICATION USAGE** section has been updated to describe certain programming considerations associated with the X/OPEN UNIX functions.

signgam

Issue 4: The header **<math.h>** is added to the **SYNOPSIS** section.

sigpause( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *signal*( ).

sigpending( )

Issue 4: No changes are made to this interface in Issue 4.

sigprocmask( )

Issue 4: The type of argument *set* is changed from **sigset_t*** to **const sigset_t***.

The description in Issue 4 is updated to indicate that signals can also be generated by the *raise*( ) function.

sigrelse( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *signal*( ).

sigsetjmp( )

Issue 4: Issue 4 states that *sigsetjmp*( ) is a macro or a function; Issue 3 states that it is a macro. Warnings are added about the suppression of a *sigsetjmp*( ) macro definition.

The description in Issue 4 is more complete than in Issue 3, though the essential capability of the interface remains unchanged. For example:

- Issue 4 describes the accessibility of objects after a *siglongjmp*( ) call. This text is imported from the entry for *longjmp*( ).
- Issue 4 describes the contexts in which calls to *sigsetjmp*( ) are valid. This text is imported from *setjmp*( ).

sigstack( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *sigstack*( ) function allows the calling process to indicate to the system an area of its address space to be used for processing signals received by the process.

This function has been marked **TO BE WITHDRAWN**. A portable application, whether being written or rewritten, should use *sigaltstack*( ) instead of *sigstack*( ). The **APPLICATION USAGE** section further describes caveats to be considered due to differences in historical implementations.

sigsuspend( )

Issue 4: The type of argument *sigmask* is changed from **sigset_t*** to **const sigset_t***.

sin( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] may be set if the value of x is NaN, or x is ±Inf. Issue 3 defines ±HUGE_VAL rather than ±Inf.

- [ERANGE] may be returned if the result underflows. Issue 3 defines that [ERANGE] may be returned if the magnitude of *x* is such that total or partial loss of significance results.

sinh( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] is returned if the result would cause overflow. Issue 3 defined this as a ''may fail'' condition.
- [ERANGE] may be returned if the result would cause underflow. Issue 3 did not mention this condition at all.

sleep( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated to indicate possible interactions with the *setitimer*( ), *ualarm*( ) and *usleep*( ) functions.

sprintf( )

Issue 4: Refer to *fprintf*( ).

sqrt( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [EDOM] is returned if the value of *x* is negative. This is defined as a ''may fail'' in Issue 3.

srand( )

Issue 4: The argument *seed* is explicitly defined as **unsigned int**.

srand48( )

Issue 4: The header **<stdlib.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

Also in the **SYNOPSIS** section, **long** is expanded to **long int**.

srandom( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *initstate*( ).

sscanf( )

Issue 4: Refer to *fscanf*( ).

stat( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows this header to be included.

The type of argument *path* is changed from **char*** to **const char***.

The description is updated in Issue 4:

- to indicate the purpose of this interface
- to define the contents of **stat** structure members
- the words ‘‘extended security controls’’ are replaced by ‘‘additional or alternate file access control mechanisms’’.

These changes are made for alignment with the POSIX-1 standard and do not imply any functional change in the interface.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [EIO] will be returned if a physical I/O error has occurred, and [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution. A second (optional) [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link, and [EOVERFLOW] has been added to indicate that a value to be stored in a member of the **stat** structure would cause overflow.

statvfs( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *fstatvfs*( ).

stdin

Issue 4: No changes are made to this interface in Issue 4.

step( )

Issue 4: The header **<regexp.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of arguments *string* and *expbuf* are changed from **char*** to **const char***.

The interface is marked TO BE WITHDRAWN, because improved capability is now provided by interfaces introduced for alignment with the XPG4 documentation (see *regcomp*( )).

strcasecmp( )

Issue 4, Version 2:
The *strcasecmp*( ) and *strncasecmp*( ) functions perform case-insensitive string comparisons, and were first introduced in Issue 4, Version 2 (X/OPEN UNIX).

strcat( )

Issue 4: The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strchr( )

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strcmp( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strcoll( )

Issue 4: Because this function is defined in the ISO C standard, it is no longer marked as an extension.

The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

Issue 3 contained a paragraph describing how the sign of the return value can be determined. This statement is misleading as it is only allowed for character ordering, whereas Issue 4 permits ordering by either string or character.

strcpy( )

Issue 4: The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strcspn( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strdup( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *strdup*( ) function returns a pointer to a new string that is a duplicate of the string pointed to by *s1*. The returned pointer can be passed to *free*( ).

strerror( )

Issue 4: Because the *strerror*( ) function is defined in the ISO C standard, it is no longer marked as an extension.

The description in Issue 4 is changed to indicate that the contents of error message strings produced by this function are locale-dependent, as determined by the setting of category LC_MESSAGES. There are a couple of points to note about this statement:

- Issue 3 states that the message strings are language-dependent, which is not completely accurate as language is only one element of a locale definition.
- LC_MESSAGES is a new category in Issue 4. Issue 3 does not define which category is used to define error strings, nor indeed if they are localised at all.

strfmon( )

Issue 4: New function in the Enhanced Internationalisation feature group in Issue 4.

The *strfmon*( ) provides an interface similar to *strftime*( ) for the formatting of monetary values. It is a higher-level and generally more usable interface than *localeconv*, for example:

```
double  val;
char    buf[MAXSIZE];
...
setlocale (LC_ALL, "");
...
if (strfmon (buf, MAXSIZE, "%#5=*n", val))
    printf (
        catgets (catd, setn, msgx,
            "amount = %s\n"),
        buf);
else
    printf (
        catgets (catd, setn, msgy,
            "value too large \n"))
...
```

formats a monetary value with five digits to the left of the radix character, filled with asterisks if necessary, producing output of the form:

```
$**123.45
```

strftime( )

Issue 4: The type of argument *format* is changed from **char*** to **const char***, and the type of argument *timptr* is changed from **struct tm*** to **const struct tm***.

In the description of the %Z conversion specification, the words ‘‘or abbreviation’’ are added to indicate that *strftime*( ) does not necessarily return a full timezone name.

Various extensions are made to this interface in Issue 4, namely:

- The description defines a new set of modified conversion specifiers, which provide access to alternative formats or specifications (for example, alternative date and time representation).
- %C, %e, %u and %V are added to the list of valid conversion specifications.

While not defined in Issue 3, these features are additive and should not invalidate existing Issue 3-compliant applications.

The text in Issue 4 is changed to make it clear when this function uses byte values rather than (possibly multi-byte) character values.

strlen( )

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function works in units of bytes rather than (possibly multi-byte) characters.

strncasecmp( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *strcasecmp*( ).

strncat( )

Issue 4: The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strncmp( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strncpy( )

Issue 4: The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strpbrk( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function works in units of bytes rather than (possibly multi-byte) characters.

strptime( )

Issue 4: New function in the Enhanced Internationalisation feature group in Issue 4.

The *strptime*( ) function provides the opposite capability to *strftime*( ) and converts a character string containing the date and time to values stored in a **tm** structure, under the control of a format string. For example:

```
...
if (strptime (argv[1], "%A %B %d %T", &tm) == NULL) {
      puts ( catgets (catd, NL_SETD, ERROR,
          "incorrect date format");
      exit (2)
}
...
```

which expects date and time as a string of the form:

```
Sun Mar 31 14:23:00
```

strrchr( )

Issue 4: The type of argument *s* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function works in units of bytes rather than (possibly multi-byte) characters.

strspn( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function works in units of bytes rather than (possibly multi-byte) characters.

strstr( )

Issue 4: The type of arguments *s1* and *s2* are changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function works in units of bytes rather than (possibly multi-byte) characters.

strtod( )

Issue 4: Because *strtod* is defined in the ISO C standard, it is no longer marked as an extension in the XSH specification.

The type of argument *str* is changed from **char*** to **const char***, and the name of the second argument is changed from *ptr* to *endptr*.

The description is changed to make it clear when the function manipulates bytes and when it manipulates characters.

strtok( )

Issue 4: Because *strtok*( ) is defined in the ISO C standard, it is no longer marked as an extension in the XSH specification.

The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear that this function manipulates bytes rather than (possibly multi-byte) characters.

strtol( )

Issue 4: Because *strtol*( ) is defined in the ISO C standard, it is no longer marked as an extension in the XSH specification.

The type of argument *str* is changed from **char*** to **const char***, and the name of the second argument is changed from *ptr* to *endptr*.

Issue 4 states that LONG_MAX or LONG_MIN is returned if the converted value is too large or too small. In Issue 3, it is implementation-dependent what would happen in these circumstances.

The description is changed to make it clear when the function manipulates bytes and when it manipulates characters.

strtoul( )

Issue 4: New function in Issue 4.

The *stroul*( ) function is similar to *strtol*, except that it converts a string to a type **unsigned long** value.

strxfrm( )

Issue 4: Because *strxfrm*( ) is defined in the ISO C standard, it is no longer marked as an extension in the XSH specification.

The type of argument *s2* is changed from **char*** to **const char***.

The description in Issue 4 is changed to make it clear when the function manipulates byte values and when it manipulates characters.

swab( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *src* is changed from **char*** to **const void***, *dest* is changed from **char*** to **void***, and *nbytes* is changed from **int** to **size_t**.

The description now states that copying between overlapping objects results in undefined behaviour. This caveat is not stated in Issue 3.

swapcontext( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *makecontext*( ).

symlink( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *symlink*( ) function creates a symbolic link, whose name is the pathname pointed to by *path2*, and whose contents are the pathname pointed to by *path1*.

Like a hard link, a symbolic link allows a file to have multiple logical names. The presence of a hard link guarantees the existence of a file, even after the original name has been removed. A symbolic link provides no such guarantee. Symbolic links, however, can cross file system boundaries.

sync( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *sync*( ) function causes all information in memory that updates file systems to be scheduled for writing out to all file systems.

sysconf( )

Issue 4: The type of the function return value is expanded to **long int**.

The following variables are added to the list of configurable system variables that can be interrogated by the *sysconf*( ) function:

BC_BASE_MAX
BC_DIM_MAX
BC_SCALE_MAX
BC_STRING_MAX
COLL_WEIGHTS_MAX
EXPR_NEST_MAX
LINE_MAX
_POSIX2_C_BIND
_POSIX2_C_DEV
_POSIX2_C_VERSION
_POSIX2_CHAR_TERM
_POSIX2_FORT_DEV
_POSIX2_FORT_RUN
_POSIX2_LOCALEDEF
_POSIX2_SW_DEV
_POSIX2_UPE
_POSIX2_VERSION
RE_DUP_MAX
STREAM_MAX
TZNAME_MAX
_XOPEN_VERSION
_XOPEN_CRYPT
_XOPEN_ENH_I18N
_XOPEN_SHM

Issue 4, Version 2:
For X/OPEN UNIX conformance, the ATEXIT_MAX, IOV_MAX, PAGESIZE, PAGE_SIZE and _XOPEN_UNIX variables were added to the list of configurable system values that can be determined by calling *sysconf*( ).

syslog( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *closelog*( ).

system( )

Issue 4: Because *system*( ) is defined in the ISO C standard, it is no longer marked as an extension in the XSH specification.

The name of the argument is changed from *string* to *command*, and its type is changed from **char*** to **const char***.

The **DESCRIPTION** and **RETURN VAUE** sections are completely replaced to bring them in line with the POSIX-2 standard. They still describe essentially the same capability as Issue 3, albeit that the definition is more complete.

The **ERRORS** section is changed to indicate that the *system*( ) function may return error values described for *fork*( ).

tan( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] is set if the value to be returned would overflow. In Issue 3, this is a ''may fail'' condition.

tanh( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten for alignment with the ISO C standard and to rationalise error handling in the mathematics functions. Specifically:

- Issue 3 states that [ERANGE] might be returned if the correct result would underflow. This condition is not defined in Issue 4.

tcdrain( )

Issue 4: Because job control is defined as mandatory in Issue 4, any attempts to use the *tcdrain*( ) function from a process that is a member of a background process group, causes the process group to be sent a SIGTTOU signal. In Issue 3, this behaviour is only defined if _POSIX_JOB_CONTROL is defined.

Issue 4 defines that *tcdrain*( ) may fail with [EIO] set, but adds the following caveat:

> ''In the POSIX-1 standard, the possibility of an [EIO] error occurring is described in Section 7.1.1.4, Terminal Access Control, but it is not mentioned in the *tcdrain*( ) interface definition. It has become clear that this omission is unintended, so it is likely that the [EIO] error will be reclassified as a ''will fail'' when the POSIX-1 standard is next updated.''

tcflow( )

Issue 4: The descriptions of TCIOFF and TCION are reworded, indicating the intended consequences of transmitting stop and start characters. Issue 3 implies that these consequences are guaranteed.

Because job control is defined as mandatory in Issue 4, any attempts to use the *tcflow*( ) function from a process that is a member of a background process group causes the process group to be sent a SIGTTOU signal. In Issue 3, this behaviour is only defined if _POSIX_JOB_CONTROL is defined.

Issue 4 defines that *tcflow*( ) may fail with [EIO] set, but adds the following caveat:

> ''In the POSIX-1 standard, the possibility of an [EIO] error occurring is described in Section 7.1.1.4, Terminal Access Control, but it is not mentioned in the *tcdrain*( ) interface definition. It has become clear that this omission was unintended, so it is likely that the [EIO] error will be re-classified as a ''will fail'' when the POSIX-1 standard is next updated.''

tcflush( )

Issue 4: The **DESCRIPTION** section is modified to indicate that the flush operation only results if the call to *tcflush*( ) is successful.

Because job control is defined as mandatory in Issue 4, any attempts to use the *tcflush*( ) function from a process that is a member of a background process group, cause the process group to be sent a SIGTTOU signal. In Issue 3, this behaviour is only defined if _POSIX_JOB_CONTROL is defined.

Issue 4 defines that *tcflush*( ) may fail with [EIO] set, but adds the following caveat:

> ''In the POSIX-1 standard, the possibility of an [EIO] error occurring is described in Section 7.1.1.4, Terminal Access Control, but it is not mentioned in the *tcdrain*( ) interface definition. It has become clear that this omission was unintended, so it is likely that the [EIO] error will be re-classified as a ''will fail'' when the POSIX-1 standard is next updated.''

tcgetattr( )

Issue 4: No changes are made to this interface in Issue 4, although the description is expanded to more fully describe baud rates and how they are obtained.

tcgetpgrp( )

Issue 4: The function is no longer marked OPTIONAL FUNCTIONALITY. This is because job control is defined as mandatory for Issue 4 conforming implementations. The [ENOSYS] error is removed from the **ERRORS** section.

The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

Issue 4 states that if there is no foreground process, a value greater than 1 is returned that does not match the process group ID of any existing process. This capability is not defined in Issue 3.

tcgetsid( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *tcgetsid*( ) function obtains the process group ID for the session for which the terminal specified by *fildes* is the controlling terminal.

tcsendbreak( )

Issue 4: Because job control is defined as mandatory in Issue 4, any attempts to use the *tcsendbreak*( ) function from a process that is a member of a background process group cause the process group to be sent a SIGTTOU signal. In Issue 3, this behaviour is only defined if _POSIX_JOB_CONTROL is defined.

Issue 4 defines that *tcsendbreak*( ) may fail with [EIO] set, but adds the following caveat:

> ''In the POSIX-1 standard, the possibility of an [EIO] error occurring is described in Section 7.1.1.4, Terminal Access Control, but it is not mentioned in the *tcdrain*( ) interface definition. It has become clear that this omission was unintended, so it is likely that the [EIO] error will be re-classified as a ''will fail'' when the POSIX-1 standard is next updated.''

tcsetattr( )

Issue 4: The type of argument *termios_p* is changed from **struct termios*** to **const struct termios***.

Three extra paragraphs are added to the description in Issue 4 describing:

- the requirements and results of calls that are partially successful
- constraints on usage of the **termios** structure pointed to by *termios_p*
- implementation restrictions on changes to the **termios** structure.

These changes are too extensive to repeat here and application developers are advised to study them before use of the interface. Applications compliant with Issue 3 should continue to work without change.

The [EINTR] error is added to the **ERRORS** section. The description of the [EINVAL] error is extended to indicate that this error will also be returned if an attempt is made to set an unsupported value in the **termios** structure.

Because job control is defined as mandatory in Issue 4, any attempts to use the *tcsetattr*( ) function from a process that is a member of a background process group, will cause the process group to be sent a SIGTTOU signal. In Issue 3, this behaviour is only defined if _POSIX_JOB_CONTROL is defined.

Issue 4 defines that *tcsetattr*( ) may fail with [EIO] set, but adds the following caveat:

> ''In the POSIX-1 standard, the possibility of an [EIO] error occurring is described in Section 7.1.1.4, Terminal Access Control, but it is not mentioned in the *tcdrain*( ) interface definition. It has become clear that this omission was unintended, so it is likely that the [EIO] error will be re-classified as a ''will fail'' when the POSIX-1 standard is next updated.''

tcsetpgrp()

Issue 4: The function is no longer marked OPTIONAL FUNCTIONALITY. This is because job control is defined as mandatory for XSI-conformant implementations. The [ENOSYS] error is removed from the **ERRORS** section.

The header **<unistd.h>** is added to the **SYNOPSIS** section. The header **<sys/types.h>** is no longer required explicitly; this header is optional on XSI-conformant systems, although the POSIX-1 standard shows **<sys/types.h>** to be included.

Text referring to how this interface should be supported on implementations that do not support job control is removed.

tdelete()

Issue 4: The function return value is changed from **char*** to **void***, the type of argument *key* is changed from **char*** to **const void***, *rootp* is changed from **char**** to **void**, and arguments to the *compar*( ) function are formally defined.

telldir()

Issue 4: The header **<sys/types.h>** is removed from the **SYNOPSIS** section.

The function return value is expanded to **long int**.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to indicate that a call to *telldir*( ) immediately following a call to *seekdir*( ) returns the *loc* value passed to the *seekdir*( ) call.

tempnam()

Issue 4: The type of arguments *dir* and *pfx* are changed from **char*** to **const char***.

The description is changed to indicate that *pfx* is treated as a string of bytes and not as a string of (possibly multi-byte) characters.

tfind()

Issue 4: The function return value is changed from **char*** to **void***, the type of argument *key* is changed from **char*** to **const void***, *rootp* is changed from **char**** to **void* const***, and arguments to the *compar*( ) function are formally defined.

time()

Issue 4: The **RETURN VALUE** section is updated to indicate that **(time_t)**–1 is returned on error.

times()

Issue 4: All references to the constant {CLK_TCK} are removed. Issue 4 applications are advised to use:

```
sysconf(_SC_CLK_TCK);
```

to determine the number of clock ticks per second.

The **RETURN VALUE** section is updated to indicate that **(clock_t)**–1 is returned on error.

timezone

Issue 4: Issue 4 defines that this variable contains the difference between UTC (Co-ordinated Universal Time) and the local standard time. Issue 3 states that the difference is indicated between GMT and the local standard time.

The type of *timezone* is expanded to **extern long int**.

tmpfile( )

Issue 4: The argument list is explicitly defined as **void**.

The [EINTR] error is moved to the ‘‘will fail’’ part of the **ERRORS** section, and [EACCES], [ENOTDIR] and [EROFS] are removed.

tmpnam( )

Issue 4: No changes are made to this interface in Issue 4.

toascii( )

Issue 4: No changes are made to this interface in Issue 4.

_tolower( )

Issue 4: No changes are made to this interface in Issue 4.

tolower( )

Issue 4: No functional changes are made to this interface in Issue 4, but the following should be noted:

- Reference to ‘‘shift information’’ is replaced by ‘‘character-type information’’.
- The last paragraph of the description in Issue 3, describing default actions if no case conversion information is defined, is removed. The POSIX locale is now defined separately elsewhere in the XSH specification.

_toupper( )

Issue 4: No changes are made to this interface in Issue 4.

toupper( )

Issue 4: No functional changes are made to this interface in Issue 4, but the following should be noted:

- Reference to ‘‘shift information’’ is replaced by ‘‘character-type information’’.
- The last paragraph of the description in Issue 3, describing default actions if no case conversion information is defined, is removed. The POSIX locale is now defined separately elsewhere in the XSH specification.

towlower( )

Issue 4: New function in Issue 4 (WPI).

The *towlower*( ) function is similar to *tolower*( ), except that it accepts a wide character as input and returns the lower-case equivalent (if any), again encoded as a wide character.

towupper( )

Issue 4: New function in Issue 4 (WPI).

The *towupper*( ) function is similar to *toupper*( ), except that it accepts a wide character as input and returns the upper-case equivalent (if any), again encoded as a wide character.

truncate( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

Refer to *ftruncate*( ).

tsearch( )

Issue 4: The type of argument *key* in the definition of the *tsearch*( ) function is changed from **void*** to **const void***. The definitions of other functions have changed as indicated on their respective entries.

Various minor wording changes are made in the description to improve clarity and accuracy. In particular, additional notes have been added about constraints on the first argument to the *twalk*( ) function.

ttyname( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

ttyslot( )

Issue 4, Version 2:
This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ttyslot*( ) function returns the index of the current user's entry in the user accounting database.

twalk( )

Issue 4: The type of argument *root* is changed from **char*** to **const void***, and the argument list to the *action*( ) function is formally defined.

tzname

Issue 4: The header **<time.h>** is added to the **SYNOPSIS** section.

tzset( )

Issue 4: The argument list is explicitly defined as **void**.

ualarm( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *ualarm*( ) function causes the SIGALRM signal to be generated for the calling process after the number of real-time microseconds specified in the *useconds* argument has elapsed.

ulimit( )

Issue 4: The use of **long** is replaced by **long int** in the **SYNOPSIS** section and the description.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to revise the discussion of UL_GETFSIZE and UL_SETFSIZE to distinguish between the soft and the hard file size limit of the process.

umask( )

Issue 4: The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included.

Issue 4 defines that unspecified bits in the file mode creation mask can be restored by a subsequent call to *umask*( ), using a *cmask* value returned from an earlier call. This is not specified in Issue 3.

uname( )

Issue 4: Issue 4 states that the format of information stored in the **utsname** structure by the *uname*( ) function is unspecified. This has always been the case, even though it is not stated explicitly in earlier versions of the XSH specification.

Issue 4 also states that –1 is returned on error, although it does not define what errors occur. It is conceivable, because *name* is a pointer to a structure, that some implementations may check the pointer value and return [EFAULT] if it is invalid.

ungetc( )

Issue 4: The *fsetpos*( ) function is added to the list of file-positioning functions in the description. Also Issue 4 states that the file-position indicator is decremented by each successful call to the *ungetc*( ) function, although note that XSI-conformant systems do not distinguish between text and binary streams. Previous issues state that the disposition of this indicator is unspecified.

The description is changed to make it clear that the *ungetc*( ) function manipulates bytes rather than (possibly multi-byte) characters.

Issue 4 states that the file-position indicator is decremented by each successful call to the *ungetc*( ) function, except that if its value is zero before the call, it is indeterminate afterwards. The description in Issue 3 is slightly different and states:

- that the value of the file-position indicator is unspecified after a successful call to the *ungetc*( ) function until all pushed-back characters are read or discarded
- that in the case that *stream* is *stdin*, and the stream is buffered, one character can be pushed back onto the buffer without a previous read statement.

Because of the ambiguity in the Issue 3 definition of this function, applications could not rely on the setting of the file-position indicator, so the changes listed above should not affect portability. Also there is no loss of capability implied by the Issue 4 definition of *ungetc*( ).

ungetwc( )

Issue 4: New function in Issue 4 (WPI).

The *ungetwc*( ) function is functionally similar to *ungetc*( ), except that it pushes a wide character back onto *stream*. The only other thing to note about this function is that the file position indicator may be decremented by more than one if the pushed back wide character converts to a multi-byte sequence.

unlink( )

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *path* is changed from **char*** to **const char***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to account for the case if *path* is a symbolic link.

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and [EPERM] and [EACCESS] have been added to indicate a permission check failure when operating on directories with S_ISVTX set. A second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

unlockpt( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *unlockpt*( ) function unlocks the slave pseudo-terminal device associated with the master to which *fildes* refers.

usleep( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *usleep*( ) function suspends the current process from execution for the number of microseconds specified by the *useconds* argument.

The *usleep*( ) function is included for historical usage. The *setitimer*( ) function is preferred over this function.

utime( )

Issue 4:

The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included.

The type of argument *path* is changed from **char*** to **const char***, and *times* is changed from **struct utimbuf*** to **const struct utimbuf***.

Since the behaviour associated with {_POSIX_NO_TRUNC} is supported on all Issue 4 systems, [ENAMETOOLONG] is always returned if a pathname component is larger than {NAME_MAX}. On Issue 3 systems, the pathname can be truncated by the system, so the condition is silently ignored.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **ERRORS** section has been updated to indicate that [ELOOP] will be returned if too many symbolic links are encountered during pathname resolution, and a second [ENAMETOOLONG] condition is defined that may report excessive length of an intermediate result of pathname resolution of a symbolic link.

utimes( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *utimes*( ) function sets the access and modification times of the file pointed to by the *path* argument to the value of the *times* argument. The *utimes*( ) function allows time specifications accurate to the microsecond.

valloc( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *valloc*( ) function has the same effect as *malloc*( ), except that the allocated memory will be aligned to a multiple of the value returned by *sysconf(_SC_PAGESIZE)*.

Applications should avoid using *valloc*( ), and use *malloc*( ) or *mmap*( ) instead.

vfork()

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *vfork*( ) function has the same effect as *fork*( ), except that the child process can share code and data with the calling process (its parent). This speeds cloning activity significantly at the risk of the integrity of the parent process if *vfork*( ) is misused. The use of *vfork*( ) for any purpose except as a prelude to an immediate call to an *exec*( ) function, or *_exit*( ) is not advised.

vfprintf()

Issue 4:

Because these functions are specified in the ISO C standard, they are no longer marked as extensions in the XSH specification.

The type of argument *format* is changed from **char*** to **const char***.

Issue 3 states that these functions are called with an argument list as defined in **<varargs.h>**, whereas Issue 4 states that this definition is found in **<stdarg.h>**. The **<varargs.h>** header is still defined in Issue 4, but it is marked TO BE WITHDRAWN. Thus applications that depend on this header still work correctly on Issue 4 systems, but they may not work on later systems.

Note that this is one XSI function that requires two headers.

wait( )

Issue 4:

The header **<sys/types.h>** is no longer required explicitly. This header is optional on XSI-conformant systems, although the POSIX-1 standard shows it to be included.

Text describing conditions under which zero is returned when WNOHUNG is set in *options* is modified in Issue 4. Specifically, the following caveats are added:

> ''..., it has at least one child process specified by *pid* for which status is not available, and status is not available for any process specified by *pid*, ...''

This is a correction to the description and does not affect application portability.

The words ''If the implementation supports job control'' are removed from the description of WUNTRACED. This is because job control is defined as mandatory for Issue 4 conforming implementations.

Issue 4, Version 2:

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to add the WCONTINUED options flag, and the WIFCONTINUED(*stat_val*) macro, to explain the implications of setting the SA_NOCLDWAIT signal flag, or setting SIGCHLD to SIG_IGN, and to explain what macros return non-zero values in which cases.

wait3( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *wait3*( ) function allows the calling process to obtain status information for specified child processes. It can return resource usage information for the child process as well.

waitid( )

Issue 4, Version 2:

This interface was first introduced in Issue 4, Version 2 (X/OPEN UNIX).

The *waitid*( ) function suspends the calling process until one of its children changes state.

wcscat( )

Issue 4: New function in Issue 4 (WPI).

The *wcscat*( ) function is similar to *strcat*( ), except that it concatenates two wide-character strings.

wcschr( )

Issue 4: New function in Issue 4 (WPI).

The *wcschr*( ) function is similar to *strchr*( ), except that it scans a wide-character string for the first occurrence of a specified wide-character code.

wcscmp( )

Issue 4: New function in Issue 4 (WPI).

The *wcscmp*( ) function is similar to *strcmp*( ), except that it compares two wide-character strings.

wcscoll( )

Issue 4: New function in the Enhanced Internationalisation feature group in Issue 4.

The *wcscoll*( ) function is similar to *strcoll*( ), except that it compares two wide-character strings using collating information in the program locale.

wcscpy( )

Issue 4: New function in Issue 4 (WPI).

The *wcscpy*( ) function is similar to *strcpy*( ), except that it copies one wide-character string to another wide character string.

wcscspn( )

Issue 4: New function in Issue 4 (WPI).

The *wcscspn*( ) function is similar to *strcspn*( ), except that it computes the length of the maximum initial segment of a wide character string that consists entirely of wide-character codes not from another wide-character string.

wcsftime( )

Issue 4: New function in the Enhanced Internationalisation feature group in Issue 4.

The *wcsftime*( ) function is similar to *strftime*( ), except that it converts date and time to a wide-character string. This function differs from the approved MSE working draft in that the type of the format argument is **wchar_t *** in the MSE working draft and **char *** in the XSH specification.

Note: This function is aligned with the ISO SC22/WG14/N104 draft of the ISO Working Paper SC22/WG14/N205 dated 31 March 1992. The type of the *format* argument may be changed.

wcslen( )

Issue 4: New function in Issue 4 (WPI).

The *wcslen*( ) function is similar to *strlen*( ), except that it computes the number of wide-character codes in a wide-character string.

wcsncat( )

Issue 4: New function in Issue 4 (WPI).

The *wcsncat*( ) function is similar to *strncat*( ), except that it concatenates part of two wide-character strings.

wcsncmp( )

Issue 4: New function in Issue 4 (WPI).

The *wcsncmp*( ) function is similar to *strncmp*( ), except that it compares part of two wide-character strings.

wcsncpy( )

Issue 4: New function in Issue 4 (WPI).

The *wcsncpy*( ) function is similar to *strncpy*( ), except that it copies part of two wide-character strings.

wcspbrk( )

Issue 4: New function in Issue 4 (WPI).

The *wcspbrk*( ) function is similar to *strpbrk*( ), except that it scans one wide-character string for wide-character code specified in a second string.

wcsrchr( )

Issue 4: New function in Issue 4 (WPI).

The *wcsrchr*( ) function is similar to *strrchr*( ), except that it scans a wide-character string for the last occurrence of a specified wide-character code.

wcsspn( )

Issue 4: New function in Issue 4 (WPI).

The *wcsspn*( ) function is similar to *strspn*( ), except that it computes the length of the maximum initial segment of a wide-character string that consists entirely of wide-character codes from another wide-character string.

wcstod( )

Issue 4: New function in Issue 4 (WPI).

The *wcstod*( ) function is similar to *strtod*( ), except that it converts a wide-character string to a double precision number.

wcstok( )

Issue 4: New function in Issue 4 (WPI).

The *wcstok*( ) function is similar to *strtok*( ), except that it splits a wide-character string into tokens. This function differs from the approved MSE working draft in that there is an additional argument in the MSE working draft.

wcstol( )

Issue 4: New function in Issue 4 (WPI).

The *wcstol*( ) function is similar to *strtol*( ), except that it converts a wide-character string to a long integer.

wcstombs( )

Issue 4: New function in Issue 4.

This function converts a wide-character string of type **wchar_t*** to a character string of type **char***. This is normally done before outputting such a string to an external storage medium.

Note: Explicit conversion is not required if wide-character strings are output via the WPI standard I/O functions.

wcstoul( )

Issue 4: New function in Issue 4 (WPI).

The *wcstoul*( ) function is similar to *strtoul*( ), except that it converts a wide-character string to an **unsigned long** integer.

wcswcs( )

Issue 4: New function in Issue 4 (WPI).

The *wcswcs*( ) function is similar to *strstr*( ), except that it locates a wide-character substring in another wide-character string. This function differs from the approved MSE working draft in that the name was changed to *wcsstr*( ) in the MSE working draft.

wcswidth( )

Issue 4: New function in Issue 4 (WPI).

The *wcswidth*( ) function computes the number of column positions in a wide-character string. This is different from *wcslen*( ), which returns the number of wide-character codes in a wide-character string, irrespective of how many column positions might be required to display the string.

wcsxfrm( )

Issue 4: New function in the Enhanced Internationalisation feature group in Issue 4.

The *wcsxfrm*( ) function is similar to *strxfrm*( ), except that it transforms two wide-character strings.

wctomb( )

Issue 4: New function in Issue 4 (WPI).

This function converts a wide-character code of type **wchar_t** to a byte string of type **char***. This is normally done before outputting such a code to an external storage medium.

Note: Explicit conversion is not required if wide-character codes are output via the WPI standard IO functions.

wctype( )

Issue 4: New function in Issue 4 (WPI).

This interface is used with *iswctype*( ) to provide classification facilities for all character classes in a locale, including those for which there is no direct *isw<class>*( ) function. The interface accepts the name of a character class on input, and returns a value of type **wctype_t** for use in subsequent calls to *iswctype*( ).

For example, the *iswalpha*( ) function could be implemented as:

```
iswalpha(wc)   iswctype(wc, wctype("alpha"))
```

More usefully, this interface in combination with *iswctype*( ) provides a mechanism for identifying character classes other than those defined for English language use.

wcwidth()

Issue 4: New function in Issue 4 (WPI).

This function is similar to *wcswidth*(), except that it computes the number of column positions required to display a single wide character.

wordexp()

Issue 4: New function in the POSIX.2 C-language Binding feature group in Issue 4.

The *wordexp*() function performs word expansions the same as those performed by the shell if *words* were part of a command line representing the arguments to a utility. For example:

```
wordexp ("Those 'who can,' do", &pwordexp, 0);
```

would result in three words being returned; that is:

```
Those
who can,
do
```

Note: *wordexp*() obeys the shell's rules for quoting, parameter and command substitution, and tilde, pathname and arithmetic expansion.

The structure pointed to by the second argument contains at least the fields:

we_wordc Count of words matched.

we_wordv Pointer to list of expanded words.

we_offs Slots to reserve at the beginning of **we_wordv**.

The third argument is a bit-significant list of flags that can be used to control the specific operation of *wordexp*().

write()

Issue 4: The header **<unistd.h>**, which contains the function prototype for this interface, is added to the **SYNOPSIS** section.

The type of argument *buf* is changed from **char*** to **const void***, and the type of argument *nbyte* is changed from **unsigned** to **size_t**.

Various minor changes are made to the description of *write*() in Issue 4, as follows:

- Issue 4 states that writing at end-of-file is atomic; that is, if the O_APPEND flag is set, no intervening file modification occurs between changing the file offset and the write operation. This behaviour is also true in Issue 3, even though it is not stated explicitly.
- Issue 4 states that the result is implementation-dependent if *nbyte* is greater than {SSIZE_MAX}. This limit is defined by the constant {INT_MAX} in Issue 3. This is unlikely to have any practical implications for applications as the minimum acceptable value for both constants is defined to be 32767.
- Issue 4 defines the consequences of activities following a call to *write*(). This is clarification of the text and does not imply any functional differences to Issue 3.
- The text describing operations on pipes or FIFOs when O_NONBLOCK is set is restructured to improve clarity.

- Issue 4 states that if *write*( ) is interrupted by a signal after it has successfully written some data, it returns the number of bytes written. In Issue 3 it is optional whether *write*( ) returns the number of bytes written, or whether it returns –1 with *errno* set to [EINTR].

Issue 4, Version 2:

The *writev*( ) function has been added to the **SYNOPSIS** section, for X/OPEN UNIX conformance, and an operational description for *writev*( ) has been added to the **DESCRIPTION** section, along with appropriate changes to the **RETURN VALUE** and **ERRORS** sections.

The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to describe writing data to STREAMS files.

The **ERRORS** section was reorganised to describe errors that apply generally to both *write*( ) and *writev*( ), and errors that apply specifically to *writev*( ). The [EIO], [ERANGE] and [EINVAL] errors are also added, and a second [ENXIO] error return described.

y0( )

Issue 4: References to the *matherr*( ) function are removed.

The **RETURN VALUE** and **ERRORS** sections are substantially rewritten to rationalise error handling in the mathematics functions. Specifically:

- [ERANGE] may also be set if *x* is zero, or the correct result overflows or underflows.

Chapter 8

Headers

8.1 Header and Name Space Rules

Issue 4, Version 2 of the XSH specification specifies headers from a number of formal standards and specifications, and must handle the name space issues carefully. Each specification builds on its predecessors. XPG3 Base includes everything from the POSIX-1 standard, and the POSIX-1 standard includes everything from the ISO C standard. XPG4 Base includes additionally the POSIX-2 standard and the MSE working draft. Issue 4, Version 2 adds the X/Open UNIX Extension feature group, specifying core APIs of 4.3BSD, OSF AES and SVID3.

8.1.1 ISO C Headers

The ISO C standard describes general rules for implementations *versus* applications with respect to headers and name space.

Headers:

1. Headers need not be regular text files and the characters between the < and > need not name a source file.
2. Headers should be self-sufficient such that any standard header does not need any other standard header to be included using **#include** before or after it (and thus headers can be included in any order).
3. Headers should be idempotent, such that any standard header can be included any number of times without causing problems.
4. Headers can provide macro versions of functions which should behave in a semantically identical fashion. By undefining such a macro, a regular declaration of the function is made visible.
5. Headers can assume that they are included outside of any file scope declaration or definition.
6. Headers can assume that they are included prior to the first use of its functions, objects or macros.

With the exception of **<assert.h>**, the ISO C standard headers are both self-sufficient and idempotent.

The ISO C standard specifies the rules for its 15 standard headers. Items 2 through 6 listed above for headers are requirements for these.

<assert.h>	**<locale.h>**	**<stddef.h>**
<ctype.h>	**<math.h>**	**<stdio.h>**
<errno.h>	**<setjmp.h>**	**<stdlib.h>**
<float.h>	**<signal.h>**	**<string.h>**
<limits.h>	**<stdarg.h>**	**<time.h>**

A compilation system can provide more headers, but a strictly conforming ISO C program can only use these.

Name space:

1. _[_A-Z][0-9a-z_A-Z]* are reserved for implementation use anywhere.
2. _[0-9a-z_A-Z]* are reserved for implementation use as identifiers with external linkage and in any header as a tag or an ordinary identifier with file scope.

8.1.2 POSIX-1 Headers

Other standards disagree slightly regarding the contents of some of these headers. For example, the POSIX-1 standard specifies that *fdopen*( ) is declared in **<stdio.h>**. To allow these two standards to coexist, the POSIX-1 standard requires that the macro _POSIX_SOURCE be defined using **#define** before a standard header is included. The POSIX-specific names are then declared and no more than those permitted by POSIX.

The POSIX-1 standard specifies general rules for name space control through ‘‘feature test macros’’.

1. Feature test macros have names that match _[_A-Z][0-9a-z_A-Z]*. (Thus use of such names take advantage of the loophole left in the ISO C standard.) They almost always match _[A-Z]+_SOURCE. The feature test macro for the POSIX-1 standard is _POSIX_SOURCE.
2. If the feature test macro associated with a particular header is defined prior to its inclusion, the header must obey the name space rules of the matching specification.
3. If such a header is included without the feature test macro being defined, it has no restrictions (at least with respect to the associated specification).

The POSIX-1 standard specifies general rules for its headers.

1. [a-z_A-Z][0-9a-z_A-Z]*_t are reserved for all POSIX.1 headers. It is unclear whether this reservation is intended to be in effect for those headers that are covered by the ISO C standard. This makes the assumption that it does apply.
2. Each function must be declared with a prototype in at least one header (ISO C standard implementations).
3. The default location for this declaration is **<unistd.h>**.

The POSIX-1 standard specifies the rules for its 12 standard headers and subsumes the 15 headers from the ISO C standard, giving additional rules to 6 of them.

<dirent.h>	**<sys/stat.h>**	**<sys/wait.h>**
<fcntl.h>	**<sys/times.h>**	**<termios.h>**
<grp.h>	**<sys/types.h>**	**<unistd.h>**
<pwd.h>	**<sys/utsname.h>**	**<utime.h>**

The 6 modified ISO C standard headers are:

<errno.h>	**<setjmp.h>**	**<stdio.h>**
<limits.h>	**<signal.h>**	**<time.h>**

8.1.3 XPG Headers

The XPG3 XSI specification set subsumes the POSIX-1 standard and specifies rules for 11 more headers.

<ftw.h>	**<search.h>**	**<sys/shm.h>**
<langinfo.h>	**<sys/ipc.h>**	**<ulimit.h>**
<nl_types.h>	**<sys/msg.h>**	**<varargs.h>**
<regexp.h>	**<sys/sem.h>**	

A feature test macro named _XOPEN_SOURCE provides for all the X/Open extensions to the ISO C standard, in a similar way to the _POSIX_SOURCE macro. The definition of _XOPEN_SOURCE subsumes the use of _POSIX_SOURCE and _POSIX_C_SOURCE, ensuring the appropriate POSIX name space is also exposed. If the system defines _XOPEN_UNIX, then the application needs to further define _XOPEN_SOURCE_EXTENDED to completely expose the correct name space.

The XPG4 XSI specification set additionally specifies that no more than one header need be included to use any one function.

The XPG4 XSI specification set subsumes the POSIX-1 standard, and adds in the MSE working draft and POSIX-2 standard headers.

<cpio.h>	**<iconv.h>**	**<tar.h>**
<fnmatch.h>	**<monetary.h>**	**<wchar.h>**
<glob.h>	**<regex.h>**	**<wordexp.h>**

Finally, the XSH specification, Issue 4, Version 2 is a superset of Issue 4 of the XSH specification, adding in the X/Open UNIX Extension feature group. This adds the following headers:

<fmtmsg.h>	**<re_comp.h>**	**<sys/resource.h>**	**<sys/uio.h>**
<libgen.h>	**<strings.h>**	**<sys/statvfs.h>**	**<sys/un.h>**
<ndbm.h>	**<stropts.h>**	**<sys/time.h>**	**<syslog.h>**
<poll.h>	**<sys/mman.h>**	**<sys/timeb.h>**	**<ucontext.h>**
<utmpx.h>			

8.2 Names Safe to Use

The rules regarding when certain names are reserved are complicated. There are, however, four fairly simple rules which if followed avoid collisions with any ISO C reserved names:

1. Use **#include** to include all system headers at the top of source files.
2. Do not define or declare any names that begin with an underscore.
3. Use an underscore or a capital letter somewhere within the first few characters of all file scope tags and regular names.

   Note: Beware of the **va_** prefix found in **stdarg.h**.
4. Use a digit or a non-capital letter somewhere within the first few characters of all macro names.

   Note: Almost all names beginning with an E are reserved if **<errno.h>** is included.

Most implementations continue to add names to the standard headers. The _POSIX_SOURCE macro is principally used to remove the added declarations.

8.3 Header Migration Information

This section contains information for each header defined in the XSH specification. Each section identifies changes made to the interface in Issue 4 (if any), complete with examples where appropriate. Only changes that might affect an application programmer are identified.

<assert.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

<cpio.h>

Issue 4: This entry is moved from Supplementary Definitions, Issue 3.

Issue 4, Version 2:
Descriptions for C_ISLNK and C_ISSOCK were provided. These were previously listed as reserved.

<ctype.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

<dirent.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

A statement is added to the description indicating that the internal format of directories is unspecified. Also in the description of the **d_name** field, the text is changed to indicate bytes rather than (possibly multi-byte) characters.

<errno.h>

Issue 4: The [EILSEQ] error is added and marked as an extension.

The [ENOTBLK] error is withdrawn.

Issue 4, Version 2:
The following symbolic constants were added for X/OPEN UNIX conformance:

EADDRINUSE	EADDRNOTAVAIL	EAFNOSUPPORT	EALREADY
EBADMSG	ECONNABORTED	ECONNREFUSED	ECONNRESET
EDESTADDRREQ	EDQUOT	EHOSTUNREACH	EINPROGRESS
EISCONN	ELOOP	EMSGSIZE	EMULTIHOP
ENETDOWN	ENETUNREACH	ENOBUFS	ENODATA
ENOLINK	ENOPROTOOPT	ENOSR	ENOSTR
ENOTCONN	ENOTSOCK	EOPNOTSUPP	EOVERFLOW
EPROTO	EPROTONOSUPPORT	EPROTOTYPE	ESTALE
ETIME	ETIMEDOUT	EWOULDBLOCK	

<fcntl.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

A reference to **<unistd.h>** is added for the definition of *l_whence*, SEEK_SET, SEEK_CUR and SEEK_END, and marked as an extension.

A reference to **<sys/stat.h>** is added for the symbolic names of file modes used as values of **mode_t**, and marked as an extension.

A reference to **<sys/types.h>** is added for the definition of **mode_t**, **off_t** and **pid_t**, and marked as an extension.

A warning is added indicating that inclusion of **<fcntl.h>** may also make visible all symbols from **<sys/stat.h>** and **<unistd.h>**. This is marked as an extension.

<float.h>

Issue 4: New header in Issue 4.

<fmtmsg.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<fnmatch.h>

Issue 4: New header in Issue 4.

<ftw.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

A reference to **<sys/stat.h>** is added for the definition of the **stat** structure, the symbolic names for **st_mode** and the file type test macros.

A warning is added indicating that inclusion of **<ftw.h>** may also make visible all symbols from **<sys/stat.h>**.

Issue 4, Version 2:
The **DESCRIPTION** section has been updated for X/OPEN UNIX conformance to define the **FTW** structure, to include the declaration of *nftw*( ), to add the FTW_SL and FTW_SLN macros to support symbolic links, and to define macros for the fourth argument to *nftw*( ).

<glob.h>

Issue 4: New header in Issue 4.

<grp.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

A reference to **<sys/types.h>** is added for the definition of **gid_t** and marked as an extension.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *endgrent*( ), *getgrent*( ) and *setgrent*( ) functions are declared in this header.

<iconv.h>

Issue 4: New header in Issue 4.

<langinfo.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The constants CODESET, T_FMT_AMPM, ERA, ERA_D_FMT, ALT_DIGITS, YESEXPR and NOEXPR are added.

Reference to the Gregorian calendar is removed.

The constants YESSTR and NOSTR are now defined as belonging to category LC_MESSAGES. In previous issues they are defined as constants in category LC_ALL.

A warning is added indicating that inclusion of **<langinfo.h>** may also make visible all symbols from **<nl_types.h>**.

The **APPLICATION USAGE** section is expanded to recommend use of the *localeconv* ( ) function.

<libgen.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X / OPEN UNIX conformance.

<limits.h>

Issue 4: This entry is largely restructured to improve symbol grouping. A great many symbols, too numerous to mention, are also added for alignment with the POSIX-2 standard. Additional changes are made as follows:

- The constants INT_MIN, LONG_MIN and SHRT_MIN are changed from values ending in 8 to values ending in 7.
- The DBL_DIG, DBL_MAX, FLT_DIG and FLT_MAX symbols are marked both as extensions and TO BE WITHDRAWN.
- The LONG_BIT and WORD_BIT symbols are marked as extensions.
- The DBL_MIN and FLT_MIN symbols are withdrawn.
- Text introducing numerical limits now indicates that they are constant expressions suitable for use in **#if** preprocessing directives.
- The minimum acceptable value for NGROUPS_MAX is changed from _POSIX_NGROUPS_MAX to 8, indicating that supplementary groups are supported on all conforming implementations. This change is made for alignment with the FIPS requirements.
- A sentence is added to the **DESCRIPTION** section indicating that names beginning with _POSIX can be found in **<unistd.h>**.
- The PASS_MAX and TMP_MAX symbols are marked TO BE WITHDRAWN.
- Use of the terms ‘‘bytes’’ and ‘‘characters’’ is rationalised to make it clear when the description is referring to either single-byte values or possibly multi-byte characters.

- CHARCLASS_NAME_MAX is added to the list of invariant values and marked as an extension.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the following **Run-time Invariant Values** have been added: ATEXIT_MAX, IOV_MAX, PAGESIZE and PAGE_SIZE. The following **Minimum Value** has been added: _XOPEN_IOV_MAX.

<locale.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The definition of **struct lconv** is added.

A reference to **<stddef.h>** is added for the definition of NULL.

<math.h>

Issue 4: The description of HUGE_VAL is changed to indicate that this value is not necessarily representable as a **float**.

The function declarations in this header are expanded to full ISO C prototypes.

The functions declared in this header are subdivided into those defined in the ISO C standard, and those defined only by X/Open. Functions in the latter group are marked as extensions, as is the external variable *signgam*.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the following functions are now declared in this header: *acosh*( ), *asinh*( ), *atanh*( ), *cbrt*( ), *expm1*( ), *ilogb*( ), *log1p*( ), *logb*( ), *nextafter*( ), *remainder*( ), *rint*( ), *scalb*( ).

<monetary.h>

Issue 4: New header in Issue 4.

<ndbm.h>

Issue 4, Version 2:

New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<nl_types.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

<poll.h>

Issue 4, Version 2:

New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<pwd.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definitions of **gid_t** and **uid_t**. This is marked as an extension.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *endpwent*( ), *getpwent*( ) and *setpwent*( ) functions are declared in this header.

<regex.h>

Issue 4: New header in Issue 4.

<re_comp.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<regexp.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The interface is marked TO BE WITHDRAWN.

<search.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definition of **size_t**.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *insque*( ) and *remque*( ) are added to the list of functions declared in this header.

<setjmp.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The **DESCRIPTION** section is changed to indicate that all functions in this header can also be declared as macros.

The types **jmp_buf** and **sigjmp_buf** are explicitly defined as array types. While this is almost certainly the case on systems compliant with Issue 3 as well, it is not mandated in the Issue 3 interface definition.

Issue 4, Version 2:
For X/OPEN UNIX conformance, *_longjmp*( ) and *_setjmp*( ) are added to the list of functions declared in this header.

<signal.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The description in Issue 4 is changed:

- to define the type **sig_atomic_t**
- to define the syntax of signal names and functions
- to define that SIGFPE is no longer limited to floating-point exceptions, but covers all arithmetic errors.

The *raise*() function is added to the list of functions declared in this header.

A reference to **<sys/types.h>** is added for the definition of **pid_t**. This is marked as an extension.

In the list of signals starting with SIGCHLD, the statement ''but a system not supporting the job control option is not obliged to support the capability of these signals'' is removed. This is because job control is defined as mandatory on XSI-conformant implementations.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the following changes have been made to **<signal.h>**:

- The SIGBUS, SIGPOLL, SIGPROF, SIGSYS, SIGTRAP, SIGURG, SIGVTALRM, SIGXCPU and SIGXFSZ signals have been added as supported signals.
- The *sa_sigaction* member is added to the **sigaction** structure, with the added note that the storage used by *sa_handler* and *sa_sigaction* may overlap.
- The SA_ONSTACK, SA_RESETHAND, SA_RESTART, SA_SIGINFO, SA_NOCLDWAIT, SS_ONSTACK, SS_DISABLE, MINSIGSTKSZ and SIGSTKSZ constants are defined. The **stack_t**, **siginfo_t** and **ucontext_t** types are defined. The **sigstack** structure is defined.
- A set of macros is defined as signal-specific reasons why a signal was generated as values for **si_code**.
- The *bsd_signal*(), *killpg*(), *sigaltstack*(), *sighold*(), *sigignore*(), *siginterrupt*(), *sigpause*(), *sigrelse*(), *sigset*() and *sigstack*() functions are added to the list of functions declared in this header.

<stdarg.h>

Issue 4: New header in Issue 4.

<stddef.h>

Issue 4: New header in Issue 4.

<stdio.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The description in Issue 4 is restructured to group lists of macro names according to how they are defined by an implementation (for example, whether they are integral constant expressions, pointer constants, string constants, and so on).

The constant FILENAME_MAX is added to the list of integral constant expressions. The text of FOPEN_MAX is also changed for consistency with the ISO C standard.

The data type **fpos_t** is added. This is referred to in the **APPLICATION USAGE** section in Issue 3.

The functions *fgetpos*( ) and *fsetpos*( ) are added to the list of functions declared in this header.

The constant L_cuserid, and the external variables *optarg*, *opterr*, *optind* and *optopt* are marked as extensions and TO BE WITHDRAWN.

The P_tmpdir constant is moved from the **APPLICATION USAGE** section to the **DESCRIPTION** section and marked as an extension.

The **va_list** type, and a reference to the definition of **size_t** in **<stddef.h>**, are added and marked as extensions.

The *cuserid*( ) and *getopt*( ) functions are marked TO BE WITHDRAWN.

The external variables *optarg*, *opterr*, *optind* and *optopt* are defined, marked as extensions and marked TO BE WITHDRAWN.

A warning is added indicating that inclusion of **<stdio.h>** may also make visible all symbols from **<stddef.h>**.

<stdlib.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The minimum maximum value of RAND_MAX is defined to be at least 32767.

The name MB_CUR_MAX is added to the list of macro names defined in this header, while **div_t** and **ldiv_t** are added to the list of defined types.

The names *atexit*( ), *div*( ), *labs*( ), *ldiv*( ), *mblen*( ), *mbstowcs*( ), *mbtowc*( ), *strtoul*( ), *wcstombs*( ) and *wctomb*( ) are added to the list of functions declared in this header.

A reference is added to **<stddef.h>** and **<wchar.h>** for the definition of **size_t**.

A reference is added to **<sys/wait.h>** for definitions of the symbolic names and macros defined for decoding the return value from the *system*( ) function.

The names *drand48*( ), *erand48*( ), *jrand48*( ), *lcong48*( ), *lrand48*( ), *mrand48*( ), *nrand48*( ), *putenv*( ), *seed48*( ), *setkey*( ) and *srand48*( ) are added to the list of functions declared in this header and marked as extensions.

A warning is added indicating that inclusion of **<stdlib.h>** may also make visible all symbols from **<stddef.h>**, **<limits.h>**, **<math.h>** and **<wchar.h>**.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the following functions have been added to the list of functions declared in this header: *a64l*( ), *ecvt*( ), *fcvt*( ), *gcvt*( ), *getsubopt*( ),

grantpt( ), *initstate*( ), *l64a*( ), *mktemp*( ), *mkstemp*( ), *ptsname*( ), *random*( ), *realpath*( ), *setstate*( ), *srandom*( ), *ttyslot*( ), *unlockpt*( ), *valloc*( ).

<string.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The name *memmove*( ) is added to the list of functions declared in this header.

A reference is added to **<stddef.h>** for the definition of **size_t**.

A warning is added indicating that inclusion of **<string.h>** may also make visible all symbols from **<stddef.h>**.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *strdup*( ) function is added to the list of functions declared in this header.

<strings.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<stropts.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/ipc.h>

Issue 4: Reference to the header **<sys/types.h>** is added for the definitions of **uid_t**, **gid_t** and **mode_t**.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *ftok*( ) function is added to the list of functions declared in this header.

<sys/mman.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/msg.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and **size_t**.

A statement is added indicating that all symbolic constants in **<sys/ipc.h>** are defined when this header is included.

<sys/resource.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/sem.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and **size_t**( ).

A statement is added indicating that all symbolic constants in **<sys/ipc.h>** are defined when this header is included.

<sys/shm.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definitions of **pid_t**, **time_t**, **key_t** and **size_t**.

A statement is added indicating that all symbolic constants in **<sys/ipc.h>** are defined when this header is included.

<sys/stat.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The description in Issue 4 is expanded to indicate:

- how files are uniquely identified within the system
- that times are given in units of seconds since the Epoch
- rules governing the definition and use of the file mode bits
- usage of the file type test macros.

Reference to the header **<sys/types.h>** is added for the definitions of **dev_t**, **ino_t**, **mode_t**, **nlink_t**, **uid_t**, **gid_t**, **off_t** and **time_t**( ). This has been marked as an extension.

Reference to the S_IREAD, S_IWRITE and S_IEXEC file modes is removed.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the following additions have been made to this header:

- The **st_blksize** and **st_blocks** members have been added to the **stat** structure.
- The S_IFLNK value of S_IFMT is defined.
- The S_ISVTX file mode bit and S_ISLNK file type test macro are defined.
- The *fchmod*( ), *lstat*( ) and *mknod*( ) functions are added to the list of functions declared in this header.

<sys/statvfs.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/time.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/timeb.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/times.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definitions of **clock_t**.

This issue states that the *times*( ) function can also be defined as a macro.

<sys/types.h>

Issue 4: The data type *ssize_t* is added.

The description is expanded to indicate the required arithmetic types.

Issue 4, Version 2:
The **id_t** and **useconds_t** types are defined for X/OPEN UNIX conformance.

<sys/uio.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<sys/utsname.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The description is changed to state that although the structure members are character arrays of unspecified size, they are terminated by a null byte (not a null character as stated in previous issues).

This issue states that the *uname*( ) function can also be defined as a macro.

<sys/wait.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definition of **pid_t** and marked as an extension.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the following additions have been made:

- The WIFCONTINUED macro, the list of symbolic constants for use with the *options* argument of *waitid*( ), and the description of the **idtype_t** enumeration type have all been added.

- The *wait3*( ) and *waitid*( ) functions are added to the list of functions declared in this header.
- A statement has been added indicating that inclusion of this header may also make visible symbols from **<signal.h>** and **<sys/resource.h>**.

<syslog.h>

Issue 4, Version 2:
New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<tar.h>

Issue 4: This entry is moved from Supplementary Definitions, Issue 3.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the significance of SYMTYPE as the value of *typeflag*, and of TSVTX as the value of *mode* have been explained.

<termios.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Some minor rewording of the description is done to align the text more exactly with the POSIX-1 standard. No functional differences are implied by these changes.

The list of mask name symbols for the *c_oflag* field have all been marked as extensions, with the one exception of OPOST.

The following words are removed from the description of the **c_cc** array:

> ''Implementations that do not support the job control option, may ignore the SUSP character value in the **c_cc** array indexed by the VSUSP subscript.''

This is because job control is defined as mandatory for XSI-conformant implementations.

The mask name symbols IUCLC and OLCUC are marked TO BE WITHDRAWN.

Issue 4, Version 2:
For X/OPEN UNIX conformance, the *tcgetsid*( ) function is added to the list of functions declared in this header.

<time.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

The range of *tm_min* is changed from [0,61] to [0,59].

Possible settings of **tm_isdst** and their meanings are added.

The names *clock*( ) and *difftime*( ) are added to the list of functions declared in this header.

The symbolic name CLK_TCK is marked TO BE WITHDRAWN. Warnings about its use have also been added.

Reference to the header **<sys/types.h>** is added for the definitions of **clock_t**, **size_t** and **time_t**( ).

References to CLK_TCK are changed to CLOCKS_PER_SEC in part of the **DESCRIPTION** section. The fact that CLOCKS_PER_SEC is always one millionth of a second on XSI-conformant systems is marked as an extension.

External declarations for *daylight*, *timezone* and *tzname* are added. The first two are marked as extensions.

The function *strptime*( ) is added to the list of functions declared in this header.

A note about the range of **tm_sec** is added to the **APPLICATION USAGE** section.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the **<time.h>** header provides a declaration for *getdate_err*, and the *getdate*( ) function is added to the list of functions declared in this header.

<ucontext.h>

Issue 4, Version 2:

New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<ulimit.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

<unistd.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

A large number of new constants are defined for the *sysconf*( ) function, including all those with prefixes _SC_2 and _SC_BC, plus:

_SC_COLL_WEIGHTS_MAX
_SC_EXPR_NEST_MAX
_SC_LINE_MAX
_SC_RE_DUP_MAX
_SC_STREAM_MAX
_SC_TZNAME_MAX
_SC_XOPEN_CRYPT
_SC_XOPEN_ENH_I18N
_SC_XOPEN_SHM

The *confstr*( ) function is added to the list of functions declared in this header, complete with a new set of constants for alignment with the POSIX-2 standard.

This issue defines that the following symbolic constants are always defined:

_POSIX_CHOWN_RESTRICTED
_POSIX_NO_TRUNC
_POSIX_VDISABLE
_POSIX_SAVED_IDS
_POSIX_JOB_CONTROL

In Issue 3, they are only defined if the associated option is present.

The symbolic constants F_ULOCK, F_LOCK, F_TLOCK, F_TEST, GF_PATH, IF_PATH and PF_PATH are withdrawn.

The required value of _XOPEN_VERSION is defined and the constant is marked as an extension.

The constants _XOPEN_XPG2, _XOPEN_XPG3 and _XOPEN_XPG4 are added.

The constants _POSIX2_* are added.

Reference to the header **<sys/types.h>** is added for the definitions of **size_t**, **ssize_t**, **uid_t**, **gid_t off_t** and **pid_t**( ). These are marked as extensions.

The names *chroot*( ), *crypt*( ), *encrypt*( ), *fsync*( ), *getopt*( ), *getpass*( ), *nice*( ) and *swab*( ) are added to the list of functions declared in this header. With the exception of *getopt*( ), these are all marked as extensions.

Issue 4, Version 2:

For X/OPEN UNIX conformance, the following additions have been made to this header:

- The feature group constant _XOPEN_UNIX is defined.
- The *sysconf*( ) symbolic constants _SC_ATEXIT_MAX, _SC_IOV_MAX, _SC_PAGESIZE and _SC_PAGE_SIZE are defined.
- The following functions are added to the list of functions declared in this header: *brk*( ), *fchown*( ), *fchdir*( ), *ftruncate*( ), *getdtablesize*( ), *gethostid*( ), *getpagesize*( ), *getpgid*( ), *getsid*( ), *getwd*( ), *lchown*( ), *lockf*( ), *readlink*( ), *sbrk*( ), *setpgrp*( ), *setregid*( ), *setreuid*( ), *symlink*( ), *sync*( ), *truncate*( ), *ualarm*( ), *usleep*( ), *vfork*( ).
- The symbolic constants F_ULOCK, F_LOCK, F_TLOCK and F_TEST are added.

<utime.h>

Issue 4: The function declarations in this header are expanded to full ISO C prototypes.

Reference to the header **<sys/types.h>** is added for the definition of **time_t**. This is marked as an extension.

<utmpx.h>

Issue 4, Version 2:

New header in Issue 4, Version 2 for X/OPEN UNIX conformance.

<varargs.h>

Issue 4: The interface is marked TO BE WITHDRAWN.

The **APPLICATION USAGE** section is added, recommending use of **<stdarg.h>** in preference to this header.

<wchar.h>

Issue 4: New header in Issue 4.

<wordexp.h>

Issue 4: New header in Issue 4.

XPG3-XPG4 Base Migration Guide, Version 2

Part 4:

C-language Migration

X/Open Company Ltd.

Chapter 9

Introduction

This part describes techniques for writing C-language code that conforms to the referenced ISO C standard. It highlights differences between the C language used in Issues 3 and 4 of the Portability Guide. This chapter explains the approach used and identifies the commands used to access C compilers.

9.1 Terminology

ISO C is the term used to refer to the language specified in the referenced ISO C standard. ISO C is based on ANSI C, which is defined in the referenced ANSI C standard. The term *Standard C* is frequently used to encompass ISO C and ANSI C. Because the standards are technically the same, the term Standard C is not used elsewhere in this guide; the term ISO C is used. This guide assumes the reader is already familiar with the C language in use before standardisation; this is frequently termed Common Usage C or K&R C. The X/Open C language is documented in the Programming Languages, Issue 3. It is a Common Usage C. Throughout the remainder of this guide, the term X/Open C is used to describe the C language used before standardisation.

9.2 Approach

This part is useful for programmers who are:

- upgrading existing C-language code written for an Issue 3 X/Open C compiler
- writing new code which is to be compiled with an ISO C compiler.

The XCU specification describes X/Open's interface to the ISO C compiler.

9.3 Compiler

The command *c89* provides access to the language defined in the ISO C standard, whereas *cc* is the command used to access X/Open C.

A program written to X/Open C can be compiled using the *c89* command, provided that migration guidelines given in this part of the guide are followed. In this case, X/Open C function definitions are handled as functions with empty argument lists. Thus, compile-time argument checking is bypassed, but the code should still compile and run correctly.

Chapter 10

Function Prototypes

This chapter explains the implications of function prototypes, which are the most visible ISO C addition to the syntax of the C language.

10.1 Function Declarations

A function declaration defines the return type of the function, the name of the function and an optional list of arguments. The type of each argument is defined within the argument list, along with an optional argument name. For example:

```
int f(int, char);
```

This declares a function that returns a type **int**. The name of the function is *f*( ) and it takes two arguments. The first is a type **int** and the second is a type **char**.

The two arguments could also be named as in the following example:

```
int f(int i, char c);
```

The keyword **void** is used for two purposes in function prototypes. The first is for stating that a function does not return a result. For example:

```
void f(int);
```

Here, the function *f*( ) has one argument but does not return a result.

Another use of **void** is to indicate that a function does not have any arguments, for example:

```
int f(void);
```

This states that the function *f*( ) does not take any arguments, but it does return a type **int**.

Note the difference between this second use of **void** and the following example:

```
int f();
```

This indicates that nothing is specified about the arguments, not necessarily that there are no arguments.

There are two further details worth noting about function prototypes; the use of **const** and the use of ellipsis (...). They are both used in the following example:

```
int fprintf(FILE *, const char *, ...);
```

Using type **const** for the second argument indicates that it is a pointer to a type **char** string that is not changed by this function. The ellipsis indicates that there are an unknown number of additional arguments to this function (see Section 10.5 on page 248).

10.1.1 Argument Checking

The introduction of function prototypes means that the compiler can perform argument checks for each function call, in a similar fashion to the *lint* command. The compiler produces a warning indicating an improper pointer or integer combination; for example, if a function is declared as:

```
int f(int);
```

and invoked by:

```
f("Hello");
```

10.1.2 Type Conversion

The introduction of function prototypes also means that arguments to functions are automatically converted, if possible, to the type expected by the function. This conversion is the same conversion that is performed during assignment. In the following example:

```
int f(float);
   ...
int i = 32;
   ...
f(i);
```

the call to the function converts the type **int** to a type **float**.

10.2 Writing New Code

ISO C includes rules that govern the mixing of old-style and new-style function declarations. These rules recognise that there are many lines of existing C code that will not be converted to use prototypes.

When writing an entirely new program, new-style function declarations should be used in headers and new-style function declarations and definitions should be used in C source files. However, if the code is to be compiled on an X/Open C compiler, the __STDC__ macro should be used. This macro should be used with **#ifdef** or **#if** to decide which declarations to use dependent on the type of compilation system as shown in Example 10-1.

Example 10-1 Use of the __STDC__ Macro

```
#ifdef __STDC__
#if __STDC__ - 0 == 1
    void errmsg(int, ...);
    struct s *f(const char *);
    int g(void);
#else
    void errmsg();
    struct s *f();
    int g();
#endif
#else
    void errmsg();
    struct s *f();
    int g();
#endif
```

If an ISO compiler is being used, the first three declarations are used; otherwise the last three declarations are used. Note that the first three declarations say more about the functions than the last three declarations.

It is good programming practice to declare and define all functions with prototypes. An ISO C compiler issues a diagnostic message when two incompatible declarations exist for the same function. This ensures that all calls agree with the definition of the functions, eliminating some of the most common C programming mistakes.

Example 10-1 demonstrates the most robust way of handling this situation. The ISO C standard says that a conforming implementation will define __STDC__ to have the value 1. However, non-conforming implementations may still define __STDC__. For editorial purposes, the rest of the examples in this part use a simpler preprocessor test, but recognise it is also less robust.

10.3 Updating Existing Code

There are four approaches possible when updating existing code to use function prototypes:

- Recompile without making any changes.

  The compiler may provide warnings about mismatches in argument type and number.

- Add function prototypes to the headers.

  This covers all calls to global functions. Example 10-1 on page 245, which shows the use of __STDC__, is a good example of this change.

- Add function prototypes to the headers and start each source file with function prototypes for its local functions.

  This covers all calls to functions. However, this requires duplicating the interface for each local function in the source file as shown in Example 10-2.

Example 10-2 Duplicating Interface

```
#ifdef __STDC__
    static void del(struct s *);
#endif
  ...
static void del(p)
struct s *p;
{
  ...
}
```

Here the **#ifdef** ... **#endif** lines are added to declare the function *del*( ), but the function itself is unchanged.

- Change all function declarations and definitions to use function prototypes.

  This is the most extensive change. For Example 10-2 it involves changing *del*( ) to be:

```
static void del(struct s *p)
{
  ...
}
```

10.4 Mixing Old and New Styles

If a function prototype declaration is to work with an old-style function definition, they must both specify functionally identical interfaces. In ISO terminology, they must have *compatible types*.

For functions with a variable number of arguments, ISO C ellipsis notation cannot be mixed and the old-style function definitions as specified in **<varargs.h>**. However, the situation is straightforward for functions with a fixed number of arguments. The types of the arguments need to be specified *as they are passed* in X/Open C. For example, if the original definition of the function is:

```
int f(i, l)
int i;
long l;
{
  . . .
}
```

the function prototype declaration is:

```
int f(int, long);
```

However, it is also necessary to understand how the arguments are converted in X/Open C when a function is called.

In X/Open C, each argument is converted just before it is passed to the called function. The conversion is based on the default argument promotions. These specify that all integral types narrower than type **int** are promoted to type **int size**, and any type **float** argument is promoted to type **double**. These promotions are used to simplify both the compiler and the libraries.

Function prototypes are more expressive and hence the conversion can be exact. The specified argument type is the actual type of the object that is passed to the function. This means that if a function prototype is written to match an old-style function definition, the function prototype cannot include arguments of the following types:

char	**signed char**	**unsigned char**
short	**signed short**	**unsigned short**
float		

There still remain two complications with writing prototypes: **typedef** names and the promotion rules for narrow **unsigned** types.

If arguments in old-style functions are declared using **typedef** names, such as **off_t** and **ino_t**, it is important to know whether or not the **typedef** name designates a type that is affected by the default argument promotions. For these two, **off_t** is a type **long**, so it is appropriate to use in a function prototype. However, **ino_t** is a type **unsigned short**, so if it is used in a prototype, the compiler issues a diagnostic message (possibly fatal), because the old-style definition and the prototype specify different and incompatible interfaces.

The main incompatibility between X/Open C and ISO C is the promotion rule for the widening of type **unsigned char** and type **unsigned short** to a value the size of type **int**. (See Chapter 11 on page 251.) ISO C converts them to type **int** if type **int** can represent all values of the original type. Otherwise they are converted to type **unsigned int**.

The best approach is to change the old-style definition to specify either type **int** or type **unsigned int** and to use the matching type in the function prototype. If the narrower object is required, the value can be assigned to a local variable with the narrower type inside the function.

10.5 Variable Number of Arguments

The ISO C standard encourages the use of function prototypes to specify the types of a function's arguments. In order to support functions which have a variable number of arguments, such as *printf*( ), a special ellipsis terminator (...) is added to the language syntax. ISO C requires that all declarations and the definition of such a function include the ellipsis terminator. This is because an implementation might be required to do unusual things to handle a variable number of arguments.

Since there are no names for the ... part of the arguments, a special set of macros contained in the **<stdarg.h>** header gives the function access to these arguments. Pre-ISO C used similar macros contained in the **<varargs.h>** header.

Example

This example is an error-handling function *errmsg*( ). This function returns **void** and has only one fixed argument, of type **int**, that specifies details about the error message. This argument may be followed by a filename, or a line number, or both, and these are followed by a format similar to *printf*( ) and arguments that specify the text of the error message.

To allow earlier compilers to compile the example correctly, it makes extensive use of the **__STDC__** macro, which is only defined for ISO C compilation systems.

To declare the function in an appropriate header file:

```
#ifdef __STDC__
    void errmsg(int code, ...);
#else
    void errmsg();
#endif
```

Note that ISO C permits the fixed arguments to be named.

In order to define *errmsg*( ), old and new styles must be mixed, as described in Section 10.4 on page 247.

First, include different headers, dependent on the compilation system:

```
#ifdef __STDC__
#include <stdarg.h>
#else
#include <varargs.h>
#endif

#include <stdio.h>
```

Include **<stdio.h>** because the program calls the functions *fprintf*( ) and *vfprintf*( ).

For ISO C, the function's declaration is the same as the one in the header file. For X/Open C, use the identifiers **va_alist** and **va_dcl**:

```
void
#ifdef __STDC__
errmsg(int code, ...)
#else
errmsg(va_alist) va_dcl   /* note: no semicolon */
#endif
  {
    va_list ap;
    char *fmt;
    /* continued below */
```

The old-style variable argument mechanism does not allow fixed arguments to be specified. To permit access to the fixed arguments before the variable arguments, the ellipsis must have a handle. This is done by using the *va_start*( ) macro. The first argument to this macro is the handle for the Single UNIX Specification; the second is the name of the argument immediately before the ellipsis:

```
#ifdef __STDC__
    va_start(ap, code);
#else
    int code;
    va_start(ap);
    code = va_arg(ap, int);  /* extract the fixed argument */
#endif
```

The rest of the code is the same for both compilation systems:

```
if (code & FILENAME)
    (void)fprintf(stderr, "\"%s\": ", va_arg(ap, char *));
if (code & LINENUMBER)
    (void)fprintf(stderr, "%d: ", va_arg(ap, int));
if (code & WARNING)
    (void)fputs("warning: ", stderr);
fmt = va_arg(ap, char *);
(void)vfprintf(stderr, fmt, ap);
    va_end(ap);
}
```

Both the *va_arg*( ) and *va_end*( ) macros work the same for the old-style and ISO C versions. Note that *va_arg*( ) changes the value of *ap*, so the call to *vfprintf*( ) cannot be:

```
(void)vfprintf(stderr, va_arg(ap, char *), ap);
```

The macros FILENAME, LINENUMBER and WARNING must be defined in the same header in which *errmsg*( ) is declared.

A sample call to *errmsg*( ) could be:

```
errmsg(FILENAME, file, "cannot open: %s\n", argv[i]);
```

It is also possible to declare and define functions that have no fixed arguments, for example:

```
int f( . . . );
```

For these functions, use *va_start*( ) with an empty second argument, as in the following:

```
va_start(ap,);
```

Chapter 11

Promotion

This chapter considers the implications of the change in the type of *promotion* used by ISO C compilers. Promotion means changing the type of an object so that it is compatible with other objects in an arithmetic expression. The difference between the promotion used in X/Open C compilers and ISO C compilers is such that a program behaves differently, but without complaint.

11.1 Converting Types

The following expression is valid only when both objects have the same numerical type:

```
i + c
```

If *i* is a type **int** and *c* is a type **char**, *c* must be converted to an integer before the addition can be performed. This process is called *promotion*.

11.2 Background

In the referenced document by Kernighan and Ritchie the type **unsigned** specifies exactly one type. There are no types of **unsigned char**, **unsigned short** or **unsigned long**. However, most C compilers have added one or more of these. Different implementations chose different rules for type promotions when these new types were mixed with other types.

Most X/Open C compilers use the rule called *unsigned preserving*. This rule states that when an **unsigned** type needs to be promoted it is promoted to an **unsigned** type. It also states that when an **unsigned** type is mixed with a **signed** type, the result is an **unsigned** type.

The other rule is called *value preserving*. This is the rule used by ISO C. It states that promotion depends on the relative sizes of the types involved. For example, when a type **unsigned char** or **unsigned short** is promoted, the resulting type is **int** if an **int** is large enough to represent all the values of the smaller type. Otherwise the resulting type is **unsigned int**. This *value preserving* rule produces the least surprises for most expressions.

The ISO C standard comments as follows:

> QUIET CHANGE
>
> A program that depends on unsigned preserving arithmetic conversions behaves differently, probably without complaint. This is considered to be the most serious change made by the Committee to a widespread current practice.

11.3 Using a Cast

In the following code, a promotion is needed if a type **unsigned char** is smaller than a type **int**:

```
int f(void)
{
    int i = -2;
    unsigned char uc = 1;

    return (i + uc) < 17;
}
```

This is an example where the result differs dependent on the promotion rule used. An ISO C compiler may issue a warning for the (i + uc) operation to indicate that the result depends on the promotion rule being used.

The result of the addition has type **int** when using value preserving promotion and **unsigned int** for unsigned preserving promotion. However, the bit pattern is the same in both cases; for example, on a two's-complement machine:

```
   i: 111...110  (-2)
+ uc: 000...001  ( 1)
  ____________________
      111...111  (-1 or UINT_MAX)
```

This bit representation corresponds to –1 for type **int** and UINT_MAX for type **unsigned int**. For objects of type **int** a signed comparison is used and the less-than test is true. For objects of type **unsigned int** an unsigned comparison is used and the less-than test is false.

A cast should be used explicitly to define the behaviour that is required. This removes the ambiguity from the expression.

The value preserving cast is:

```
(i + (int)uc) < 17
```

The unsigned preserving cast is:

```
(i + (unsigned int)uc) < 17
```

11.4 Same Result

In the following code, two promotions are performed if types **unsigned short** and **unsigned char** are both smaller than type **int**:

```
int f(void)
{
    unsigned short us;
    unsigned char uc;

    return uc < us;
}
```

Both variables are promoted to either **int** or **unsigned int**, dependent on which promotion rule is in force. Therefore the comparison is sometimes **signed** and sometimes **unsigned**. However, the result is the same in either case, because both values are always non-negative and therefore the sense of the comparison does not matter.

11.5 Integral Constants

As with expressions, the rules for the types of certain integral constants have changed. In X/Open C, an unsuffixed decimal constant has type **int** if its value fits in a type **int**; otherwise it has type **long**. Similarly an unsuffixed octal or hexadecimal constant has type **int** if its value fits in an **unsigned int**; otherwise it has type **long**.

In ISO C, the type of the constant is the first from one of the following lists:

```
unsuffixed decimal:
    int, long, unsigned long
unsuffixed octal or hexadecimal:
    int, unsigned int, long, unsigned long
u-suffixed or U-suffixed:
    unsigned int, unsigned long
l-suffixed or L-suffixed:
    long, unsigned long
ul-suffixed or UL-suffixed:
    unsigned long
```

In the following code, the objects of type **int** have 16 bits:

```
int f(void)
{
    int i = 0;

    return i > 0xffff;
}
```

For an X/Open C compiler the constant's type is **int**, with a value of –1 on a two's-complement machine. Therefore the result of the comparison is true.

For an ISO C compiler the constant's type is **unsigned int**, with a value of 65535. Therefore the result of the comparison is false.

Again, an appropriate cast clarifies the code and silences an ISO C compiler.

For X/Open C behaviour use:

```
i > (int)0xffff
```

For ISO C behaviour use:

```
i > (unsigned int)0xffff
    or
i > 0xffffU
```

Note: The U suffix is a new feature of ISO C and probably produces an error message with older C compilers.

Chapter 12

Tokenisation and Preprocessing

This chapter discusses the operations that transform each source file from a sequence of characters into a sequence of tokens ready for parsing. These operations include recognition of white spaces and comments, bundling consecutive characters into tokens, handling preprocessing directive lines, and replacement of macros. These operations are probably the least specified part of X/Open C; their ordering was never guaranteed.

This chapter also describes some new macros and the changes to the **#define** directive.

12.1 ISO C Translation Phases

ISO C specifies eight conceptual steps, called *translation phases*. Although an implementation can merge adjacent phases together, the result must be the same as if they were distinct phases, each performed in sequence.

1. Every *trigraph sequence* in the source file is replaced by a corresponding single character.
2. Every backslash and new-line character pair is deleted.
3. The source file is converted into preprocessing tokens and sequences of white spaces. Each comment is effectively replaced by a space character.
4. Each preprocessing directive is handled and all macro invocations are replaced. Each **#include** source file is run through phases 1 to 4, recursively, before its contents replace the directive line.
5. All escape sequences in character constants and string literals are converted to their character equivalents.
6. Adjacent string literals are concatenated and wide-character string literals are concatenated.
7. Every preprocessing token is converted into a token. These tokens are then analysed for correct syntax and semantics. The compiler then uses these tokens to generate the code.
8. All external object and function references are resolved, resulting in the final program.

12.2 Trigraph Sequences

ISO C has nine trigraph sequences that were invented solely as a concession to character sets that are deficient as far as ISO C is concerned. They are three-character sequences that name a character that is not in the ISO / IEC 646: 1991 standard character set:

```
??<   {       ??(   [       ??'   ^
??>   }       ??)   ]       ??!   |
??=   #       ??-   ~       ??/   \
```

Be careful of their use; in the following example:

```
/* comment *??/
/* still a comment? */
```

the ??/ becomes a backslash. This backslash and the following newline are removed (during the second translation phase) and the resulting characters are:

```
/* comment */* still a comment? */
```

The first forward slash from the second line is the end of the comment. The next token is *.

12.3 X/Open C Translation Phases

The X / Open C compilers do not follow such a simple sequence of phases, nor do they guarantee when these steps are applied. A separate preprocessor recognises tokens and white spaces at the same time that it replaces macros and handles directive lines. The output is then completely retokenised by the compiler before it parses the language and generates the code.

The tokenisation process within the preprocessor is performed on a moment-by-moment basis and macro replacement is done as a character-based operation and not as a token-based operation. This means that tokens and white spaces could have a great deal of variation during preprocessing.

There are a number of differences that arise between these two approaches. The following four sections discuss differences in code behaviour due to line-splicing, macro replacement, string literal production and token pasting.

12.4 Logical Source Lines

In X / Open C, backslash and new-line pairs are allowed only as a means to continue a directive, a string literal or a character constant to the next line. ISO C extended the notion so that a backslash and new-line pair can continue anything to the next line. The result is called a *logical source line*. Therefore, any code that relies on the separate recognition of tokens on either side of a backslash and new-line pair does not behave as expected with an ISO C compiler.

12.5 Macro Replacement

Prior to ISO C, the macro replacement process was never described in any significant detail. This vagueness spawned many divergent implementations. Any X/Open C code that relies on anything fancier than constant replacement and simple macros is probably not truly portable. There are a number of differences between the X/Open C macro replacement implementation and the ISO C version. However, nearly all uses of macro replacement, with the exception of token pasting and string literal production, produce exactly the same series of tokens as before.

The major change to macro replacement is to require macro arguments (other than those that are operands of the macro substitution operators # and ##) to be expanded recursively prior to their substitution in the replacement token list. However, this change seldom produces an actual difference in the resulting tokens.

12.6 String Literal Production

In X/Open C, the following code:

```
#define str(a) "a!"
str(x y)
```

produced the string literal:

```
"x y!"
```

To do this, the preprocessor searched inside string literals (and character constants) for characters that looked like the macro's arguments.

ISO C recognised the importance of this feature, but could not condone operations on parts of tokens. In ISO C, all invocations of the above macro produce the string literal:

```
"a!"
```

To achieve the desired effect in ISO C, make use of the **#** macro substitution operator and the concatenation of string literals. A macro argument preceded by **#** has its corresponding unexpanded argument tokens converted into a string literal:

```
#define str(a) #a "!"
str(x y)
```

The above produces the two string literals:

```
"x y" "!"
```

which, after concatenation, produces:

```
"x y!"
```

Unfortunately, there is no direct replacement for the analogous operation for character constants. For example, in X/Open C:

```
#define CNTL(ch) (037 & 'ch')
CNTL(L)
```

would produce:

```
(037 & 'L')
```

which, in turn, evaluates to the ASCII <control>-L character.

The best solution for ISO C is to change all uses of this macro to:

```
#define CNTL(ch) (037 & (ch))
CNTL('L')
```

which is arguably more readable and more useful, as it can also be applied to expressions.

12.7 Token Pasting

In X/Open C, there are at least two ways to combine two tokens. Both invocations in the following code produce a single identifier **x1** out of the two tokens **x** and **1**:

```
#define self(a) a
#define glue(a,b) a/**/b
self(x)1
glue(x,1)
```

In ISO C, both the above invocations produce the two separate tokens **x** and **1**.

The second of the above two methods can be rewritten for ISO C by using the ## macro substitution operator. This operator removes any white space around it and concatenates the adjacent tokens to create a new token:

```
#define glue(a,b) a ## b
glue(x, 1)
```

Since ## is an actual operator, the invocation can be much freer with white space both in the definition and invocation.

There is no direct approach to produce the first of the old-style pasting schemes. However, since that scheme put the burden of the pasting at the invocation, it was used less frequently than the second form.

Remember that # and ## should only be used when __STDC__ is defined.

12.8 New Macros

ISO C has five new predefinition macros. They are:

__LINE__ This is the current line number in decimal.

__FILE__ This is the name of the file being compiled, as a string literal.

__DATE__ This is the date of compilation, as a string literal.

__TIME__ This is the time of compilation, as a string literal.

__STDC__ This is the decimal constant 1. This indicates a conforming implementation.

These macros should not be defined using **#define** or undefined using **#undef**.

12.9 Changes to #define

The operation of the **#define** directive has some additional features and some additional constraints as to how the directive can be used.

From the viewpoint of existing code, only the following constraints are relevant:

- Macros are no longer recursive. The name of the macro currently being expanded is not itself expanded if it occurs in the replacement string.
- Preprocessing directives are not allowed within a list of arguments to a macro.
- A macro name should not be redefined without an intervening **#undef** directive, unless the new definition is identical to the old one. If in doubt, the **#undef** directive should be used. This is ignored if the specified identifier is not currently defined as a macro name.

Chapter 13

Types

This chapter explains the use of the ISO C *type qualifiers* category, which contains the keywords **const** and **volatile**. It also describes the new types: **incomplete**, **compatible** and **composite**. This chapter explains where new types are permitted, and why they are useful.

13.1 Using Type Qualifiers

If an object is declared as a type qualifier **const**, the compiler may place that object in read-only memory. The program is not allowed to change its value and cannot make a further assignment to it.

Declaring an object as a type qualifier **volatile** warns the compiler that unexpected, asynchronous events may affect that object. Knowing this the compiler does not make any assumptions about that object. For example, the compiler does not try to optimise that object.

13.1.1 Type Qualifiers in Derived Types

The type qualifiers are unique in that they may modify names declared with **typedef**, such as **size_t**, and derived types, such as pointers. Derived types are those parts of the C language declarations that construct more complex types from the basic types. Pointers, arrays, functions, structures and unions are derived types. Except for functions, one or both type qualifiers can be used to change the behaviour of a derived type.

An example use of the type qualifier **const** is:

```
const int five = 5;
```

This declares and initialises an object with type **const int**, whose value is not changed by a correct program.

In fact, the order of the keywords is not significant to an ISO C compiler. For example, the declarations:

```
int const five = 5;
```

and:

```
const five = 5;
```

are identical to the above declaration in their effects.

The next example shows a type qualifier **const** applied to a pointer to an integer; that is, to a derived type:

```
int * const pci = &five;
```

Here an object with type pointer to **const int** is declared, which initially points to the previously declared object. Note that the pointer itself does not have a qualified type, but it points to a qualified type. This means that the pointer can be changed to point to another integer, but it cannot be used directly to modify the object that it points to.

The object that *pci* points to can be modified by using a cast, for example:

```
*(int *)pci = 17;
```

Note that the behaviour of this code is undefined if *pci* actually points to a type **const** object.

The next example declares a type **const** pointer to a type **int** that is defined elsewhere in the program:

```
extern int *const cpi;
```

Here, the value of *cpi* is not changed by a correct program, but it can be used to modify the object to which it points. Notice that **const** comes after the ***** in the above declaration.

The following pair of declarations produces the same effect as the previous example:

```
typedef int *INT_PTR;
extern const INT_PTR cpi;
```

The next example results in a type qualifier **const** pointer to a **const int**:

```
const int *const cpci;
```

13.1.2 The const Keyword

With hindsight, **readonly** would have been a better choice for this keyword than **const**. If **const** is read as **readonly**, declarations such as:

```
char *strcpy(char *, const char *);
```

are more easily understood. Here, the second argument is only used to read type **char** values, while the first argument overwrites the types **char** to which it points.

Furthermore, a pointer to a **const int** (as in a previous example), can still change the value of the object to which it points by use of a cast.

The two main uses for **const** are:

- to declare, possibly large, compile-time initialised tables of information as unchanging
- to specify that pointer arguments do not modify the objects to which they point.

The first use allows a program's data to be shared by other concurrent invocations of the same program. Any attempt to modify this invariant data can be detected immediately by some sort of memory protection fault.

The second use probably helps locate potential errors in programs. For example, functions that temporarily place a null character into the middle of a string are detected at compile time, if passed a pointer to a string that cannot be modified.

13.1.3 The volatile Keyword

For the programmer, **volatile** has several implications. For a compiler writer, it has one implication: no code generation shortcuts can be taken when accessing such an object.

It is a programmer's responsibility to declare every object that has the appropriate special properties, with a type qualifier **volatile**.

Four example uses of **volatile** objects are:

- an object that is a memory-mapped I/O port
- an object that is shared between multiple concurrent processes
- an object that is modified by an asynchronous signal handler

- an automatic storage duration object declared in a function that calls *setjmp*( ) and whose value is changed between the call to *setjmp*( ) and a corresponding call to *longjmp*( ).

The first three examples are all instances of an object whose value can be modified at any point during the execution of the program. For example, the seemingly infinite loop:

```
flag = 1;
while (flag)
    ;
```

is completely reasonable so long as *flag* has a type qualifier **volatile**, for example:

```
volatile int flag;
```

If the program is not going to loop forever, some asynchronous event needs to set *flag* to zero.

If *flag* does not have a type qualifier **volatile**, for example:

```
int flag;
```

the compilation system is free to change the above loop into a truly infinite loop that completely ignores the value of *flag*.

The fourth example, involving variables local to functions that call *setjmp*( ), is more involved. There are no guarantees about the values for objects matching the fourth case. To provide the most desirable behaviour, the *longjmp*( ) function would be required to examine every stack frame between the function calling *setjmp*( ) and the function calling *longjmp*( ) for saved register values. The possibility of asynchronously created stack frames makes this task even more complex. Therefore most implementors only document the undesirable side effect.

When an automatic object is declared with a type qualifier **volatile**, the compilation system knows that it has to produce code that exactly matches what the programmer wrote. Therefore, the most recent value for such an object is always in memory and as such is guaranteed to be up-to-date when *longjmp*( ) is called.

13.2 Incomplete Types

The ISO C standard uses the term *incomplete type* to formalise a fundamental aspect of the language that is often misunderstood.

The ISO C standard separates types into three distinct sets:

- function types
- object types
- incomplete types.

Function types are well known. Object types cover everything else, except when the size of the object is not known. Incomplete types refer to objects whose size is not known.

There are only three variations of incomplete types:

- void
- arrays of unspecified length
- structures and unions with unspecified content.

The type **void** differs from the other two in that it is an incomplete type that cannot be completed; it serves as a special function return and argument type.

Examples of incomplete types, include:

```
char s[];
struct a { struct b *bp; };
```

where the content of **struct** *b* is not yet listed.

13.2.1 Completing Incomplete Types

An array type is completed by specifying the array size in a following declaration which is in the same scope as the object.

An incomplete structure or union type is completed by specifying the content in a following declaration which is in the same scope.

13.2.2 Declarations

Certain declarations can use incomplete types, but others require complete object types. Those declarations that require object types are:

- array elements
- members of structures or unions
- objects local to a function.

All other declarations permit incomplete types. For example, the following are permitted:

- pointers to incomplete types
- functions returning incomplete types
- incomplete function argument types
- **typedef** names for incomplete types.

The return and argument types for functions are special. Except for **void**, an incomplete type used in such a manner must be completed by the time the function is defined or called.

Note that since array and function argument types are rewritten to be pointer types, a seemingly incomplete array argument type is not actually incomplete. The typical declaration of *argv* in *main*( ):

```
char *argv[];
```

as an unspecified length array of type **char** pointers, is rewritten to be a pointer to **char** pointers, and is therefore a complete type.

13.2.3 Expressions

Most expression operators require (complete) object types. The only three exceptions are:

- the unary **&** operator
- the first operand of the comma operator
- the second and third operands of the **?:** operator.

Most operators that accept pointer operands also permit pointers to incomplete types, unless pointer arithmetic is required.

13.2.4 Rationale

The ISO C standard would be simpler without incomplete types, but incomplete types allow forward references in structures and unions, which cannot be done any other way in C.

The only way to get two structures that have pointers to each other (without resorting to potentially invalid casts) is with incomplete types, for example:

```
struct a { struct b *bp; };
struct b { struct a *ap; };
```

13.2.5 Examples

Defining **typedef** names for incomplete structure and union types is quite useful. Complex data structures that contain pointers to each other, can be simplified by declaring a list of **typedef** statements as follows:

```
typedef struct item_tag Item;
typedef union  note_tag Note;
typedef struct list_tag List;
  ...
struct item_tag { ... };
  ...
struct list_tag {
    List  *next;
    ...
};
```

If structures and unions exist whose contents should not be available to the rest of the program, the tag can be declared, without the contents, in a header. Other parts of the program can use pointers to the incomplete structure or union without any problems, provided they do not attempt to use any of its members.

Another useful incomplete type is an external array of unspecified length. It is not always necessary to know the exact size of an array to make use of its contents, for example:

```
extern char *tzname[];
...
printf("Alternative time zone: %s\n", tzname[1]);
```

13.3 Compatible and Composite Types

With ISO C it is possible to make two declarations for the same entity that are not identical. The ISO C standard uses the term *compatible type* to denote those types that are close enough to be considered the same.

Composite types are *composite* types, the result of combining two compatible types.

13.3.1 Multiple Declarations

If a C program were only allowed one declaration for each object or function, compatible types would not be needed. However, function prototypes, separate compilation and linkage all require such a capability.

Separate translation units (that is, source files) have different rules for type compatibility from the rules for type compatibility within a single translation unit.

13.3.2 Separate Compilation

Since each compilation probably looks at different source files, most of the rules for compatible types across separate compilations are structural in nature.

- Matching scalar (that is, integral, floating and pointer) types must be compatible, as if they were in the same source file.
- Matching structures, unions and enumeration types must have the same number of members, and each matching member must have a compatible type. This includes bit-field widths.
- Matching structures and unions must have the members in the same order. The order of enumeration type members does not matter.
- Matching enumeration type members must have the same value.

An additional requirement is that the names of members, including the lack of names for unnamed members, must match for structures, unions and enumeration types. However, their respective tags need not necessarily match.

13.3.3 Single Compilation

When two declarations in the same scope describe the same object or function, the two declarations must specify compatible types. These two types are then combined into a single composite type that is compatible with the first two.

Compatible types are defined recursively. At the bottom are the type specifier keywords, along with the rules that say that type **unsigned short** is the same as type **unsigned short int** and that a type without type specifiers is the same as one with type **int**.

All other types are compatible only if the types from which they are derived are compatible. For example, two qualified types are compatible if the qualifiers, **const** and **volatile**, are identical and the unqualified base types are compatible.

13.3.4 Compatible Pointer Types

Two pointer types are compatible if the types they point to are compatible and the two pointers are identically qualified. The qualifiers for a pointer are specified after the *, so that these two declarations:

```
int *const cpi;
int *volatile vpi;
```

declare two differently qualified pointers to the same type: **int**.

13.3.5 Compatible Array Types

Two array types are compatible if their element types are compatible and, if both array types have a specified size, they match. This last part means that an incomplete array type is compatible both with another incomplete array type and an array type with a specified size.

13.3.6 Compatible Function Types

Two function types are compatible if their return types are compatible. If either or both function types have prototypes, the rules get more complicated.

For two function types with prototypes to be compatible, they also must have the same number of arguments, including use of the ellipsis notation, and the corresponding arguments must be argument-compatible.

For an old-style function definition to be compatible with a function type with a prototype, the prototype arguments must not end with an ellipsis and each of the prototype arguments must be argument-compatible with the corresponding old-style argument, after application of the default argument promotions.

For an old-style function declaration (not a definition) to be compatible with a function type with a prototype, the prototype arguments must not end with an ellipsis (...) and all of the prototype arguments must have types that are unaffected by the default argument promotions.

For two types to be argument-compatible, the types must be compatible after the top-level qualifiers, if any, are removed, and after a function or array type is converted to the appropriate pointer type.

Special Cases

There are a few surprises in this area. For example, type **signed int** behaves the same as type **int**.

Note also that each enumeration type must be compatible with some integral type. For portable programs this means that enumeration types effectively are separate types; for the most part, the ISO C standard views them in that manner.

13.3.7 Composite Type

The construction of a composite type from two compatible types is also defined recursively. The ways compatible types can differ from each other are due either to incomplete arrays or to old-style function types. As such, the simplest description of the composite type is that it is the type compatible with both of the original types, including every available array size and every available argument list from the original types.

For example, the following two function declarations are compatible:

```
int f(int (*)(),       double (*)[3]);
int f(int (*)(char *), double (*)[]);
```

The resultant composite type, which is compatible with both declarations, is:

```
int f(int (*)(char *), double (*)[3]);
```

Chapter 14

Expressions

This chapter discusses the differences between X/Open C and ISO C in the rules for grouping and evaluating expressions. It also describes some other changes to expressions.

14.1 X/Open C Rearrangement

X/Open C compilers are allowed to rearrange expressions involving adjacent operators that are mathematically associative and commutative, even in the presence of parentheses.

An operator, *op*, is commutative if:

```
a op b = b op a
```

For example, + is commutative, but – is not.

An operator, *op*, is associative if:

```
(a op b) op c = a op (b op c)
```

+ is associative, but – is not.

The following example is used to compare X/Open C and ISO C features:

```
int i, *p, f(void), g(void);
i = *++p + f() + g();
```

For X/Open C, the following three possible groupings are valid for the example. The groupings are shown using left and right braces:

```
i = { {*++p + f()} + g() };
i = { *++p + {f() + g()} };
i = { {*++p + g()} + f() };
```

Moreover, all of these groupings are valid for either of the following:

```
i = *++p + (f() + g());
i = (g() + *++p) + f();
```

If this expression is evaluated on an architecture for which either overflows cause an exception, or when the sign of the value can be inverted across an overflow, these three groupings behave differently if one of the additions overflows.

For such expressions on these architectures, the only recourse available in X/Open C is to split the expression to force a particular grouping. The following are possible rewrites that enforce the above three groupings respectively:

```
i = *++p;    i += f();    i += g();
i = f();     i += g();    i += *++p;
i = *++p;    i += g();    i += f();
```

14.2 The ISO C Rules

ISO C does not allow the rearrangement of operations that are mathematically associative and commutative but are not actually so on the target architecture. Thus, the precedence and associativity of the ISO C grammar completely describes the grouping for all expressions. All expressions must be grouped as they are parsed. This means that the example is grouped in the following way:

```
i = { {*++p + f()} + g() };
```

However, note that this still does not mean that *f*( ) must be called before *g*( ), nor that *p* must be incremented before *g*( ) is called.

It does mean, however, that in ISO C, expressions need not be split to guard against unintended overflows.

Parentheses

The ISO C standard is often erroneously described as honouring parentheses or evaluating according to parentheses; this is not true.

In ISO C, expressions are grouped according to the way they are parsed. This means that parentheses still only serve as a way of controlling how an expression is parsed. The natural precedence and associativity of expressions carry exactly the same weight as parentheses.

If the example is:

```
i = (((*(++p)) + f()) + g());
```

it makes no difference to its grouping, and thus to its evaluation.

14.3 Advantages of Rearrangement

The reasons for the X/Open C rearrangement rules are:

- The rearrangements provide many more opportunities for optimisations, such as compile-time constant folding.
- The rearrangements do not change the result of integral-typed expressions on most machines.
- Some of the operations are both mathematically and computationally commutative and associative on all machines.

In general, an implementation is permitted to deviate from the abstract machine description, provided that deviations do not change the behaviour of a valid C program. Therefore any rearrangement that has no effect on the application is permissible. The result is that this change in C does not have a significant impact on most C programmers.

14.4 Other Changes to Expressions

14.4.1 Type Float

A type **float** might not be lengthened to a type **double** when it appears in an expression. This means that expressions with operands of type **float** might be computed at a lower precision than double precision. Casts can be used to force double precision computations.

14.4.2 Pointer Subtraction

The resulting type of pointer subtraction no longer needs to be type **int**. It is a **signed** integral type given the name **ptrdiff_t** in **<stddef.h>**.

14.4.3 Empty Structure Declarations

A special meaning is given to the form:

```
struct x;
```

This hides any previous declarations of **x** (as a tag) from an outer scope. This is useful when declaring two mutually referencing structures. For example, the following code does not behave as the programmer probably intended:

```
struct x {int i;};
{
    struct y { struct x *xp; };
    struct x { struct y *yp; };
}
```

The **x** of **struct y** refers to the outer declaration of **x**. But by adding the empty structure declaration as follows:

```
struct x {int i;};
{
    struct x;
    struct y { struct x *xp; };
    struct x { struct y *yp; };
}
```

the outer declaration of **x** is not associated with the inner declaration of **y**.

14.5 Scope of Identifiers

Many X/Open C compilers promote the scope of identifiers with external linkage to file scope. However, in ISO C, identifiers declared in a nested scope are forgotten once the scope ends, for example:

```
f1()
{
    float ffun();
}

f2()
{
    int i = ffun();
}
```

Here the declaration of *ffun*( ) is not visible in *f2*( ). This means that no conversion is performed because it is assumed that *ffun*( ) returns a type **int**.

14.5.1 String Literals

In X/Open C, each string literal is required to be distinct and modifiable. (This is why they are called literals instead of constants.) ISO C allows these to be constant, permitting string literals to be placed into read-only memory, as well as encouraging the elimination of duplicate identical storage.

Code that modifies a string literal should replace the string with the name of a static array initialised to contain the string, as follows:

```
p = make_temp("tempXXX");
```

should be replaced with:

```
static char template[] = "tempXXX";
   ...
p = make_temp(template);
```

if *make_temp*( ) overwrites the string's characters, even if only temporarily.

Chapter 15

Internationalisation

This chapter describes the internationalisation features of the ISO C standard:

- multi-byte characters
- wide-character codes.

It also explains how they are used.

15.1 Multi-byte Characters

The basic difficulty in an Asian environment is that a large number of ideograms are needed for input and output. To work within the constraints of usual computer architectures, these ideograms are encoded as sequences of bytes. The associated operating systems, application programs and terminals understand these sequences as individual ideograms. These encodings also allow single-byte characters to be intermixed with ideogram byte sequences. The difficulty associated with recognising distinct ideograms depends on the encoding scheme used.

The term *multi-byte character* denotes a byte sequence that encodes a character, no matter what encoding scheme is employed. All multi-byte characters are members of the *extended character set*. A single-byte character is a special case of a multi-byte character. Essentially the only requirement placed on a multi-byte character's encoding is that it cannot use a null character as part of its encoding.

The ISO C standard specifies that program comments, string literals, character constants and header names are sequences of multi-byte characters.

15.2 Encoding Variations

An encoding scheme can be described as being *stateless* or *stateful*. In the first type, each multi-byte character is self-identifying. This means that any multi-byte character can simply be inserted between any pair of multi-byte characters. For example, one possible encoding choice is that each byte of a character that is not a single byte has the high-order bit set; the number of bytes in the character is determined by the value of the initial byte. The coded representation of each character is determined by applying the codeset encoding rules that associate a numeric value to each character.

In the second scheme, special *shift bytes* change the interpretation of subsequent bytes. An example is the method by which most character terminals get in and out of line-drawing mode. For this case, ISO C has the additional requirement that each comment, string literal, character constant and header name must both begin and end in the unshifted state.

15.3 Wide-character Codes

Some of the inconvenience of handling multi-byte characters would be eliminated if all characters were of a uniform number of bytes or bits. There can be thousands of ideograms in a character set, so a 16-bit or 32-bit sized integral value should be used to hold all its members. The full Chinese alphabet includes more than 65000 ideograms.

ISO C includes the **typedef** name **wchar_t** as the implementation-defined integral type large enough to hold all members of the extended character set, for example:

```
wchar_t c;
```

declares a wide-character code **c** which can hold any member of the extended character set.

For each wide-character code there is a corresponding multi-byte character and *vice versa*. The wide-character code that corresponds to a character of the portable character set is required to have the same value as the character of the portable character set. This requirement includes the null character. However, there is no guarantee that the value of the macro EOF can be stored in a **wchar_t**. (Just as EOF might not be representable as a **char**.)

15.4 Conversion Functions

Text recorded or generated externally to a program is normally encoded as a series of multi-byte characters. To process these characters internally, it is often convenient first to convert them to wide-character codes, process them, and convert them back to byte sequences for output. ISO C provides five library functions for this purpose:

mblen( ) Length of next multi-byte character.

mbtowc( ) Convert multi-byte character to wide-character code.

wctomb( ) Convert wide-character code to multi-byte character.

mbstowcs( ) Convert multi-byte character string to wide-character string.

wcstombs( ) Convert wide-character string to multi-byte character string.

The following is an example use of *mbtowc*( ):

```
char *mb;
wchar_t wchar;
  ...
mbtowc(&wchar, mb, strlen(mb));
```

The argument *mb* points to a multi-byte character. After the call to *mbtowc*( ), *wchar* holds the equivalent wide-character code.

For most application programs, there is no need to convert any multi-byte characters to or from wide-character codes. Utilities such as *diff*, for example, read in and write out multi-byte characters, only needing to check for an exact byte-for-byte match. More complicated programs that use regular expression pattern matching, such as *grep*, may need to understand multi-byte characters. However, this knowledge can be localised to the functions that manage the regular expressions. The utility *grep* requires no other special knowledge about how to handle multi-byte characters.

15.5 Features of the C Language

ISO C provides wide-character constants and wide string literals. These have the same form as their non-wide versions except that they are immediately prefixed by the letter L:

```
'x'           regular character constant
'¥'      regular character constant
"abc¥xyz"  regular string literal
L'x'            wide-character constant
L'¥'       wide-character constant
L"abc¥xyz" wide string literal
```

Notice that multi-byte characters are valid in both the regular and wide versions. The sequence of bytes necessary to produce the ideogram ¥ is encoding-specific, but if it consists of more than one byte, the value of the character constant '¥' is implementation-defined.

When the compilation system encounters a wide-character constant or wide-string literal, each multi-byte character is converted, as if by calling the *mbtowc*( ) function, into a wide-character code. Thus:

```
L'¥'         /* has type wchar_t */
L"abc¥xyz"   /* has type wchar_t[8] */
```

Note the length of the string is eight because it is terminated by a zero-valued **wchar_t**.

Just as regular string literals can be used as a shorthand method for character array initialisation, wide-string literals can be used to initialise **wchar_t** arrays:

```
wchar_t *wp = L"a¥z";
wchar_t x[] = L"a¥z";
wchar_t y[] = {L'a', L'¥', L'z', 0};
wchar_t z[] = {'a', L'¥', 'z', '\0'};
```

In the above example, the three arrays **x**, **y** and **z**, and the sequence pointed to by *wp*, have the same length and all are initialised with identical values.

Finally, adjacent wide-string literals are concatenated, just as with regular string literals. However, adjacent regular and wide-string literals cannot be concatenated. In fact, a compiler is not even required to complain if it does not accept such concatenations.

Chapter 16

Standard Headers and Reserved Names

This chapter presents the various categories of reserved names and some rationale for their reservations. This chapter also includes the set of naming rules for the reserved names.

16.1 Balancing Process

The ISO C standard includes library functions, macros and header files. This allows truly portable C programs to be written, but it means that there is a large set of reserved names.

For backwards compatibility, the ISO C standard includes names like **printf** and **NULL**. However, each name reserved in this way reduces the set of names available for free use in C programs.

On the other hand, before standardisation, compiler writers were free to add both new keywords to their compilers and names to headers. In this situation a program could not be guaranteed to compile on a different compiler from the one on which it was developed. In fact, a program might not even compile on the next release of the compiler on which it was developed.

Therefore, the ISO C standard permits extra names provided they conform to certain rules. The ISO C standard contains 32 keywords and almost 250 names in its headers, none of which necessarily follow any particular naming pattern.

16.2 Standard Headers

The ISO C standard provides 15 standard headers:

<assert.h> Assertion checking.

<ctype.h> Character handling.

<errno.h> Error conditions.

<float.h> Floating point limits.

<limits.h> Other data type limits.

<locale.h> Program locale.

<math.h> Mathematics.

<setjmp.h> Non-local jumps.

<signal.h> Signal handling.

<stdarg.h> Variable arguments.

<stddef.h> Common definitions.

<stdio.h> Standard input/output.

<stdlib.h> General utilities.

<string.h> String handling.

<time.h> Date and time.

A compilation system can provide more headers, but a strictly conforming ISO C program can only use these.

Other standards disagree slightly regarding the contents of some of these headers. For example, the POSIX-1 standard specifies that *fdopen*( ) is declared in **<stdio.h>**. To allow these two standards to coexist, POSIX requires that the macro _POSIX_SOURCE be defined using **#define** before a standard header is included. The POSIX-specific names are then declared and no more than those permitted by POSIX. There is also a similar macro, _XOPEN_SOURCE, that provides for all the X/Open extensions to the ISO C standard.

With the exception of **<assert.h>**, the ISO C standard headers are both self-sufficient and idempotent. Self-sufficiency means that any standard header does not need any other standard header to be included using **#include** before or after it. Idempotency means that any standard header can be included any number of times without causing problems.

16.3 Reserved Names

The ISO C standard places further restrictions on an implementation's libraries. In the past, most programmers learned not to use names like **read** and **write** for their own functions. The ISO C standard requires that only names reserved by it are introduced by references within the implementation.

The ISO C standard reserves a subset of all possible names for implementations to use as they choose. This class of names consists of identifiers that begin with an underscore and continue with either another underscore or a capital letter. That is, all names that match the following regular expression:

```
_[_A-Z][0-9_a-zA-Z]*
```

This means that, strictly speaking, a program's behaviour is undefined if it uses such an identifier.

However, undefined behaviour comes in different degrees. For an implementation that conforms to the POSIX-1 standard, if _POSIX_SOURCE is used, the program's undefined behaviour consists of certain additional names in certain headers, and the program still conforms to an accepted standard. This deliberate loophole in the ISO C standard allows implementations to conform to seemingly incompatible specifications. On the other hand, an implementation that does not conform to the POSIX-1 standard is free to behave in any manner when encountering a name such as _POSIX_SOURCE .

The ISO C standard also reserves all other names that begin with an underscore for use in header files as regular file scope identifiers and as tags for structures and unions, but not in local scopes. This means that the common existing practice of naming functions like **_filbuf** and **_doprnt** to implement hidden parts of the library is sanctioned.

ISO C reserves all names that match the following patterns. These are reserved for implementations and future standards:

<errno.h>	`E[0-9A-Z].*`
<ctype.h>	`(to|is)[a-z].*`
<locale.h>	`LC_[A-Z].*`
<math.h>	existing function names suffixed with **f** or **l**
<signal.h>	`(SIG|SIG_)[A-Z].*`
<stdlib.h>	`str[a-z].*`
<string.h>	`(str|mem|wcs)[a-z].*`

Names that begin with a capital letter are macros and are thus reserved only when the associated header is included. The rest of the names designate functions and therefore cannot be used to name global objects or functions.

16.4 Names Safe to Use

The rules regarding when certain names are reserved are complicated. There are, however, four fairly simple rules which if followed avoid collisions with any ISO C reserved names:

1. Use **#include** to include all system headers at the top of source files.
2. Do not define or declare any names that begin with an underscore.
3. Use an underscore or a capital letter somewhere within the first few characters of all file scope tags and regular names.

   Note: Beware of the **va_** prefix found in **stdarg.h**.
4. Use a digit or a non-capital letter somewhere within the first few characters of all macro names.

   Note: Almost all names beginning with an E are reserved if **<errno.h>** is included.

   Most implementations continue to add names to the standard headers. The _POSIX_SOURCE macro is principally used to remove the added declarations.

Index