*X/Open Technical Study*

# Universal Multiple-Octet Coded Character Set Coexistence and Migration

*X/Open Company Ltd.*



X/Open Technical Study

Universal Multiple-Octet Coded Character Set Coexistence and Migration

ISBN: 1-85912-031-8
X/Open Document Number: E401

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

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

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

or by Electronic Mail to:

XoSpecs@xopen.co.uk

# Contents

# 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.

  Developers who base their products on a current CAE specification can be sure that either the current specification or an upwards-compatible version of it will be referenced by a future X/Open brand (if not referenced already), and that a variety of compatible, X/Open-branded systems capable of hosting their products will be available, either immediately or in the near future.

  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.

**Corrigenda**

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

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

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

```
request Corrigenda; topic index
```

**This Document**

This document is a Technical Study (see above). It sets out a recommended strategy for a Universal Multi-Octet Coded Character Set (UCS). It explains the implications of adopting any UCS strategy and possible applications for ISO / IEC 10646 and Unicode.

This study is structured as follows:

- Chapter 1 is an introduction; it explains the background to UCS, the scope of the study, and defines terminology.
- Chapter 2 gives an overview of codesets, composite sequences, transformation formats and subsetting. It also identifies problems solved and problems raised by UCS.
- Chapter 3 covers the relationship between *byte*, *character* and the **char** data type. It also discusses interworking and application portability.
- Chapter 4 discusses the implications for X / Open Specifications.
- Chapter 5 considers how to use UCS with codeset-specific data types and interfaces, or as a process code.
- Chapter 6 identifies implementation issues.
- Chapter 7 makes specific recommendations.
- A glossary and index are provided.

**Intended Audience**

This document is intended for implementors of X / Open-compliant and POSIX-compliant systems.

**Typographical Conventions**

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for 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:
  - — 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*( ).
- Normal font is used for the names of constants and literals.
- Hexadecimal numbers are preceded by 0x.

# *Trade Marks*

LAN Manager™ is a trade mark of Microsoft Corporation.

MS-DOS® is a registered trade mark of Microsoft Corporation.

NFS® is a registered trade mark of Sun Microsystems.

UNIX® is a registered trade mark of UNIX System Laboratories, Inc. in the U.S.A. and other countries.

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

# Acknowledgements

X/Open gratefully acknowledges the work of the X/Open-UniForum Joint Internationalisation Working Group in the development of this study.

# Referenced Documents

The following documents are referenced in this study:

Arabic National Standards

CNS 11643
: Chinese National Standard 11643, 1992, the Taiwanese standard codeset.

eucJP
: Advanced Japanese EUC Code.

Indian Standard 10194

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

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

ISO 2022
: ISO 2022: 1986 Information Processing — ISO 7-bit and 8-bit Coded Character Sets — Coded Extension Techniques.

ISO 2375
: ISO 2375: 1985 Data Processing — Procedure for Registration of Escape Sequences.

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

ISO 8859
: ISO 8859-*: 1987 Information Processing — 8-bit Single-byte Coded Graphic Character Sets — Parts 1 to 10 inclusive.

ISO / IEC 10646
: ISO / IEC 10646-1: 1993, Information Technology — Universal Multiple-Octet Coded Character Set (UCS) — Part 1: Architecture and Basic Multilingual Plane.

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

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

JIS X 0208-1990
: Japanese Industrial Standard, Code of the Japanese graphic character set for information interchange.

RFC 822
: RFC 822: Standard for the Form of ARPA-Internet Text Messages (August 13, 1982) (Obsoletes RFC 733, 724).

SJIS
: Shift-JIS, the Japanese encoding method used on PCs.

Thai Industrial Standard 620

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

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

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

*Chapter 1*

# *Introduction*

This document describes major issues facing the implementors of X/Open-compliant and POSIX-compliant systems with regard to ISO/IEC 10646 support, in terms of direct support as either file codes or process codes, and in terms of interoperability.

This document considers the implications of the universal multiple-octet coded character set (UCS) in terms of the XPG4 Base profile, which covers the following components and implementations thereof:

- XPG4 Internationalised System Calls and Libraries
- XPG4 Commands and Utilities
- XPG4 C Language
- Source Code Transfer

This implicitly incorporates ISO C, ISO POSIX-1 and ISO POSIX-2, because XPG4 subsumes all these standards.

The viewpoint of this study is to consider UCS2 support in terms of existing specifications and implementations. It is not intended as a rationale either for or against UCS. ISO/IEC 10646 exists and needs to be supported, or at least accommodated, in terms of migration to and coexistence with existing systems and APIs.

This chapter provides background information and terminology.

## 1.1 Background

Information technology is evolving towards a standard for the encoding of large coded character sets that will simplify the task of writing multi-lingual applications and global information interchange. The International Organization for Standardization has developed ISO/IEC 10646 (see **Referenced Documents** on page x) to meet these requirements, while a consortium of manufacturers has independently developed Unicode. ISO/IEC 10646 is now an international standard. Unicode has been merged with ISO/IEC 10646 and is identical to the UCS-2 form of ISO/IEC 10646. Currently only the UCS-2 form of ISO/IEC 10646 has been assigned.

A number of new systems have been designed to exploit these standards (for example, Windows NT). Most early implementations of ISO/IEC 10646 will appear on personal computers (PCs). However, since host systems and PCs are increasingly required to interoperate, UCS support on PCs has profound implications for host systems, requiring them to deal with various features of ISO/IEC 10646 (for example, wide-character encodings).

There are also implications for Open Systems. X/Open-compliant and POSIX-compliant systems will need upgrading to be competitive, although this will not happen overnight. Large amounts of 8-bit data and devices will continue to exist for a considerable time to come, and systems that support ISO/IEC 10646 will have to coexist with those that only support single-byte encodings.

## 1.2 Terminology

The term UCS is used to refer to Unicode or ISO/IEC 10646 in any of its multi-octet forms. A UCS system is defined as any implementation that has the following characteristics:

- UCS or FSS-UTF (see Section 2.3 on page 6) as the system codeset.
- UCS or FSS-UTF as the text file codeset, possibly with parallel support for other file encodings.

Windows NT has these characteristics and is a prime example of a UCS system. Systems that do not have these characteristics are described as *non-UCS* systems.

UCS or FSS-UTF may or may not be supported directly on printers, displays and keyboards.

Chapter 2

# Overview

This chapter discusses codesets, composite sequences, transformation formats and subsetting; it also identifies problems solved by and raised by UCS.

## 2.1 Codesets

Commonly used codesets and encoding methods, such as ASCII, ISO 8859-1, and Japanese EUC, include characters for a single language or small group of languages. Because of this, users are limited to the languages their current codeset supports. If they use ISO 8859-1, which supports Western European languages only, it is not possible to include, say, Japanese, Greek, or Arabic characters in their text.

Some applications and users need mixtures of languages that current codesets do not support. Therefore, the goal in creating ISO/IEC 10646 was to include all characters from all significant languages: to be a universal multi-octet coded character set (UCS). The initial version of ISO/IEC 10646 contains 34,168 characters covering a long list of languages, including European, Asian ideographic, Middle Eastern, Indian and others. It also reserves 6,400 code spaces for private use.

ISO/IEC 10646 specifies the same codeset as Unicode (1.1)[1]. Unicode was developed primarily by Xerox and Apple, although other companies contributed to its design. People often use ''10646'' and ''Unicode'' interchangeably, although there are differences between the two sets. This document uses each term as appropriate.

ISO/IEC 10646 differs in some ways from codesets currently used on XPG-compliant systems. Many currently supported codesets include portable characters as single-octet entities, with code values matching either ISO/IEC 646 IRV or a form of EBCDIC. The ISO/IEC 646 IRV values are in the range 0x00-0x7f (0-127 decimal). It is common for existing software to depend on one or more ISO/IEC 646 IRV values (particularly control characters), and on the fact that such characters are always one octet each (the *de facto* standard size of a byte).

Characters in ISO/IEC 10646, in contrast, are encoded in multiple octets. Code space is organised into 128 groups of 256 planes each.

ISO/IEC 10646 allows two basic forms for characters:

1. Universal Coded Character Set 2 (UCS-2). Also known as the Basic Multilingual Plane (BMP). Characters are encoded in the lower two octets (row and cell). Predictions are that this will be the most commonly used form of ISO/IEC 10646.
2. Universal Coded Character Set-4 (UCS-4). Characters are encoded in the full four octets.

1. Unicode (1.1) is a usage profile of ISO/IEC 10646.

As an example, the following table shows encodings of uppercase **A** in ISO/IEC 646 IRV, UCS-2, and UCS-4:

| | **Binary** | **Hex** |
|---|---|---|
| ISO/IEC 646 | 01000001 | 0x41 |
| UCS-2 | 0000000001000001 | 0x0041 |
| UCS-4 | 00000000000000000000000001000001 | 0x00000041 |

At present, the repertoires of characters available in UCS-2 and UCS-4 are exactly the same, but that is expected to change over time.

Unlike the codesets and encoding methods based on ISO/IEC 646 that many implementations currently support, ISO/IEC 10646 encodes portable characters in two or four octets each. In addition, many codesets prohibit any octet of any printable character from being in the control character range (0x00-0x1f and 0x7f), but ISO/IEC 10646 makes no such restriction. Notice that in the example above, **A** includes one NULL octet in UCS-2 and three NULL octets in UCS-4.

## 2.2 Composite Sequences

In addition to the UCS-2 and UCS-4 forms, ISO/IEC 10646 also includes an encoding technique in which multiple characters can be combined to form *composite sequences*. These are already present in other standards and are designed to allow a nearly infinite variety of character combinations. Examples of other codesets using composite sequences include ISO 6937, Indian Standard 10194, Thai Industrial Standard 620, Arabic National Standards, and so on.

For example, suppose you want to encode the letter <a-acute> (lowercase a with acute accent). This letter-with-diacritic exists in ISO/IEC 10646 (code value 0x00 0xe1), but it is also possible to encode it as the plain **a** followed by an acute accent; that is:

```
+--------+--------+
|   a    |    ’   |   =    <a-acute>
+--------+--------+
```

In this case, the code value of <a-acute> is:

```
Character:              a         ’
UCS-2 Code Value:       0x00 0x61 0x03 0x01
```

The resulting composite sequence consumes four octets; that is, two for the **a** and two for the acute accent. In ISO/IEC 10646, certain characters are defined as *combining diacritical marks*, and it is permissible to combine these marks with any non-combining character. Any number of combining marks can follow a base character. For example, although the following ‘‘character’’ does not exist in any language, it is a permissible encoding in ISO/IEC 10646:

```
+--------+--------+--------+--------+-------+
|   p    |    ’   |    ~   |    ^   |   ‘   |
+--------+--------+--------+--------+-------+
```

that is, <p-acute-tilde-circumflex-grave>.

Some languages are only fully supportable in ISO/IEC 10646 through the use of combining characters. Examples include Arabic and Thai.

Although combining characters give ISO/IEC 10646 great flexibility, they also create programming challenges that do not exist in many commonly used codesets. Because not all suppliers want to revise software to handle composite character sequences, ISO/IEC 10646 has three conformance levels:

Level 1: Combining characters are not allowed.
Level 2: Combining characters are allowed for Arabic, Hebrew, Indic, and Thai scripts only.
Level 3: Combining characters are allowed with no restrictions.

Thus, with ISO/IEC 10646, it is possible for an implementation to support one or more of the following:

UCS-2, Level 1: Two-octet form, no combining characters
UCS-2, Level 2: Two-octet form, combining characters allowed with restrictions
UCS-2, Level 3: Two-octet form, combining characters allowed, no restrictions
UCS-4, Level 1: Four-octet form, no combining characters
UCS-4, Level 2: Four-octet form, combining characters allowed with restrictions
UCS-4, Level 3: Four-octet form, combining characters allowed, no restrictions

Unicode (1.1) only allows two-octet code elements and does not support all levels. It is equivalent to UCS-2, Level 3.

## 2.3 Transformation Formats

In addition to the normative forms of ISO/IEC 10646, one of the standard's informative annexes defines another form called UTF (UCS Transformation Format). Commonly known as UTF-1, this form provides some compatibility between UCS-2 or UCS-4 and ISO/IEC 646 IRV. In UTF-1 form, portable ISO/IEC 646 IRV characters shrink back from being two or four octets to being a single octet (that is, they are encoded exactly the same as ISO/IEC 646 IRV). In addition, no octets of any UTF-1 characters can be in the range 0x00-0x20 or 0x7f-0x9f, so no UTF-1 octets have the same values as control characters.

Consider these examples:

**Code values for <A>**

| | |
|---|---|
| ISO/IEC 646 IRV: | `0x41` |
| UTF-1: | `0x41` |
| UCS-2: | `0x00 0x41` |

**Code values for <a-acute>**

| | |
|---|---|
| ISO 8859-1: | `0xe1` |
| UTF-1: | `0xa0 0xe1` |
| UCS-2: | `0x00 0xe1` |

**Code values for Asian ideograph <one>**

| | |
|---|---|
| JIS X0208: | `0x30 0x6c` |
| UTF-1: | `0xf6 0x21 0xd0` |
| UCS-2: | `0x4e 0x00` |

ISO/IEC 646 IRV and UTF-1 values are identical, but other UTF-1 characters consume more octets than do those same characters in other existing standards; that is, two octets in UTF-1 versus one in ISO 8859-1, three octets in UTF-1 versus two in JIS X0208. Also note that UTF-1 does not restrict octets from having the same value as ISO/IEC 646 IRV slash (/, code value 0x2f). This means UTF-1 is not directly usable as an encoding for filenames on most UNIX-based systems, because most implementations search one 8-bit byte at a time for slashes.

Because of this and other limitations with UTF-1, a group from the X/Open-UniForum Joint Internationalisation Working Group created a second transformation format called FSS-UTF (File System Safe UCS Transformation Format) or UTF-2. In this version, the most significant bit (MSB) of an ISO/IEC 646 IRV character is 0; the MSB for all octets of all other characters is 1. The following table shows how to map UCS-* characters in a given hexadecimal range to a FSS-UTF value. In the binary values, a 0 or 1 indicates that a bit must have the listed value; an **n** indicates the bit can be either 0 or 1.

| Hex Min | Hex Max | FSS-UTF Binary Encoding |
|---|---|---|
| 00000000 | 0000007F | 0nnnnnnn |
| 00000080 | 000007FF | 110nnnnn 10nnnnnn |
| 00000800 | 0000FFFF | 1110nnnn 10nnnnnn 10nnnnnn |
| 00010000 | 001FFFFF | 11110nnn 10nnnnnn 10nnnnnn 10nnnnnn |
| 00200000 | 03FFFFFF | 111110nn 10nnnnnn 10nnnnnn 10nnnnnn 10nnnnnn |
| 04000000 | 7FFFFFFF | 1111110n 10nnnnnn 10nnnnnn 10nnnnnn 10nnnnnn 10nnnnnn |

Although this mapping shows FSS-UTF characters being up to six octets long, the current version of ISO/IEC 10646 only has UCS characters up to a hex value of FFFF. FSS-UTF characters thus would be a maximum of three octets.

FSS-UTF provides full ISO/IEC 646 IRV (ASCII) transparency (any octet that looks like ISO/IEC 646 IRV is ISO/IEC 646 IRV), and also is compatible with UNIX-based and other file systems.

## 2.4 Subsetting

Another aspect of ISO/IEC 10646 that merits mention is that the standard allows subsetting. An implementation can choose to support a subset of the character code positions within ISO/IEC 10646, and be compliant with ISO/IEC 10646. Such an implementation must identify the characters in its repertoire.

## 2.5 Problems Solved by UCS

### 2.5.1 Universal Applicability

The main advantage of fixed width ISO/IEC 10646 and Unicode encodings is that they provide a single character set, and associated set of encodings, covering scripts for most of the major languages of the world.

A script is defined as a set of characters, sharing a common nature, used by one or a set of related languages. For example, the Latin script is used to write many African, American, Asian, European and Pacific languages. A language may use more than one script. The Japanese languages, for example, are written using four scripts — Kanji (Japanese subset of Han), Katakana, Hiragana and Latin.

### 2.5.2 Simplified Internationalisation Development

UCS text is stateless in the sense that each encoded character value has a unique meaning, regardless of the order it appears in the text. In contrast, the ISO 2022 code extension method widely used in Asia to represent mixed script text, is stateful, where the interpretation of a byte depends on previous bytes in the byte stream.

UCS requires no external announcement mechanism to display the characters represented by a UCS coded character value, although note that announcement is still required for operations such as collation, character classification, and so on. The EUC encoding method, another widely used method of representing mixed script text (and the method used by SVR4-based systems), interprets a byte sequence depending on locale context; that is, the same byte sequence can represent different characters in different locales.

Because each UCS encoded character value has a fixed width, random access to a single character within a character stream is greatly simplified. Variable length encoding schemes like EUC, UTF-1 or FSS-UTF require that the byte stream is parsed from the beginning to access a character at a given position.

### 2.5.3 Interoperability

One of the apparent advantages of UCS is in the context of data interchange between systems that may be using different vendor hardware and languages. As both ISO/IEC 10646 and Unicode define the characters of most major languages within a single codeset, this perceived advantage is self-evident. However, it does raise questions about encoding methods when considered in the context of existing transfer mechanisms and interchange protocols. These issues are dealt with later in this report.

## 2.6 Problems Raised by UCS

Problems in this context are not perceived as problems with ISO/IEC 10646 or Unicode themselves, but with having to provide support for these codesets in Open System environments that conform to the current set of XPG4 interface specifications.

The following generic problems are recognised:

- Basic data mappings.
- UCS support with existing APIs.
- Data interchange between UCS and non-UCS systems.

Each of these is dealt with in the following chapters.

Chapter 3

# General Issues

This chapter explains the relationship between *byte, character* and the **char** data type. It also discusses interworking and application portability.

## 3.1 ISO C and ISO POSIX

In discussing UCS in the context of ISO C and the ISO POSIX standards, it is necessary to be very precise about usage of the terms *byte, character* and the **char** data type.

ISO C defines a *byte* as a unit of data storage large enough to hold any member of the execution environment's basic character set, where it must be possible to express the address of each individual byte of an object uniquely. ISO C further defines that a byte is composed of a contiguous sequence of bits, the length of which is implementation-defined.

A character is defined as a bit representation that fits into a byte. XPG4 modifies this definition and declares that a character is a sequence of one or more bytes representing a single graphic symbol or control code (which corresponds to the ISO C definition of a multi-byte character).

An object declared as type **char** is defined to be large enough to store any member of the basic execution character set. Its actual size on a conforming implementation is given by the constant CHAR_BIT defined in **<limits.h>**. The minimum permissible value of this constant is 8 (bits).

ISO C defines the null character as a byte value with all bits set to zero, which must be present in the basic character set and is used to terminate a character string. In a multi-byte sequence, a byte with all bits set to zero is interpreted as a null character independent of shift state.

The ISO POSIX standards defer to ISO C for a definition of the terms *null* and *byte*. Like XPG4, it defines a *character* as a sequence of one or more bytes representing a single graphic symbol.

So, within the above set of definitions, the size of both a byte and a character is implementation-defined, a zero byte value is always interpreted as a null character, and the relationship between a byte, a character (in ISO C terms) and the **char** type is a fixed one-to-one relationship. The requirement that one byte should equal one octet is not stated.

It would therefore appear possible to implement Unicode or ISO/IEC 10646 as a process code and still retain conformance to the ISO C and ISO POSIX standards, and hence XPG4. Whether it is desirable or practical to do this is another matter.

## 3.2 Interworking

The paradigm that one byte equals one octet (8-bits) is prevalent throughout existing XPG APIs, even if it is not always stated explicitly. Consider, for example, the data interchange standards, which reduce all potentially troublesome data types (integers, floating point values, and so on) to representation as strings of **char**s, in the belief that this type at least is common across all implementations. XNFS[2] is a prime example of this model.

XNFS provides data transparency between cooperating systems in a heterogeneous network by means of its eXternal Data Representation (XDR) layer. The XDR is a standard for the description and encoding of data. The following extracts from the **XNFS** specification are germane to this discussion:

> The XDR standard assumes that bytes (or octets) are portable. [From Section 3.1.]
>
> ... XDR defines a single byte order (big endian, ...) a single floating-point representation (IEEE), and so on. Any program running on any machine can use XDR to create portable data by translating its local representation to the XDR standard representations; similarly, any program running on any machine can read portable data by translating the XDR standard representations to its local equivalents. The single standard completely decouples programs that create or send portable data from those that use or receive portable data. The advent of a new machine or a new language has no effect upon the community of existing portable data creators and users. [From Section 3.1.1.]
>
> The XDR makes the following assumption: that bytes (or octets) are portable, where a byte is defined to be 8-bits of data. [From Section 3.1.2.]
>
> The representation of all items requires a multiple of four bytes (or 32 bits) of data. Four bytes is large enough to support most machine architectures efficiently, yet is small enough to keep the encoded data to a reasonable size. [From Section 3.1.3.]

Some abstract data representation needs to be defined for any application transferring data between potentially disparate computer systems. In the case of XDR, this model is determined to contain 8-bit bytes and 32-bit representations of integral values (byte encoded). Lan Manager/X (LMX)[3] defines a similar model based on the Intel chip set architecture.

XDR further defines abstract types for each of the basic data entities; that is, integer, unsigned integer, enumeration, string, structure, and so on. In the case of a string, the standard defines a composite data item made up of an unsigned integer (4-bytes, big endian), giving the maximum length of the string *n*, followed by *n* bytes of the string. This data type is used to encode such things as file and directory names, which must be 8-bit encoded character values.

It is possible to represent UCS coded character values within this model, but unfortunately any such UCS strings appearing as object names (files, directories, and so on) would probably conflict with existing file-system naming conventions. For example, even if object names were restricted to the portable filename character set defined in XPG4 (the ISO/IEC 646 set of

---

2. The X/Open Specification for file-sharing services based on the NFS architecture developed by Sun Microsystems, Inc.
3. Lan Manager/X is a systems software product that allows PCs running MS-DOS or OS/2 to use a UNIX system as a file server or host for distributed applications.

characters), but were nevertheless UCS-2, UCS-4 or Unicode encoded, every UCS character would contain at least one zero valued 8-bit byte: that is, the string terminator in non-UCS systems.

Another possible approach is to redefine the XDR protocol such that strings encode wide character values, and thus cater for both 8-bit and UCS systems. However, this is unrealistic in the short term, as it would invalidate the existing field population of systems using today's definition of XDR data types.

Note also that this problem is specific to UCS encoded data and does not apply to multi-byte codesets such as Shift-JIS. These latter codesets include the ISO/IEC 646 (ASCII) codeset within them, and obey the ISO C rule that a null character is always a string terminator regardless of shift state.

In this case, the only apparent solution for Open Systems is to translate UCS codes to FSS-UTF format prior to sending data, or after receiving data. This only applies to UCS systems running the XDR protocol; non-UCS systems would be unaffected. FSS-UTF encoded data obeys ISO C semantics and would not break host file-system semantics.

**Note:** How a UCS system would define an 8-bit data type if both bytes and the **char** type were defined as larger values is problematic.

Note also that this only applies to anything defined as an XDR **string** type. Filestore data itself is described by the **opaque** type and would not be subject to translation to FSS-UTF format, although it might be necessary to convert such data to some other form (by, for example, *iconv*( )) before it can be processed locally.

## 3.3 Application Portability

The notional data model that:

- type **char** equals 8-bits
- type **short** equals 16-bits
- type **long** equals 32-bits

is prevalent in the UNIX world and harks back to its genesis on Digital Equipment Corporation hardware. It is not defined this way in any of the standards, and no one claiming XPG4, ISO C or POSIX conformance is required to implement these mappings, but from the point of view of practical portability their support is compelling.

A large number of UNIX applications have been written assuming the above mappings and changing them can be a costly exercise. For example, hardware now commonly provides larger integer types so that a 64-bit type **long** (for example) can be supported realistically. However, types **int** and **long** are used interchangeably in many applications, which would cause the application to fail if the size of a type **long** were suddenly increased.

A common manifestation of this problem arises in *printf*( ) format strings, where integer conversion specifiers are frequently qualified incorrectly, for example:

```
long   l;
...
printf ("%d\n", l);
...
```

This statement, when executed on a machine that supports a type **long** that is longer than a type **int**, typically causes a program error. This is a trivial example of course, and is easily fixed by changing the format string to "%ld\n", but it nevertheless illustrates the point.

A similar problem arises if a type **short** is not 16 bits long, which is a common assumption made by programmers. In this case, however, problems are more subtle as a type *short* is typically promoted to **int** when passed as an argument to function calls (so the above problem with a type **long** would not be manifest). Longer types **short** tend only to cause problems when data is transferred to other systems that map them as 16-bit values; consider their use in *tar* headers, for example.

Anyone who has implemented a system whose natural data mappings are not those indicated above is aware of these problems. However, these are mere irritations when compared to the resulting traumas of changing the size of a type **char**, which the ASCII-centric world of UNIX programs casts in tablets of stone as an immutable 8-bit object.

## 3.4 Summary

The point being made is that while the standards permit bytes and type **char** to be implementation-defined, in the real world this can be an expensive diversion. The ‘‘one byte equals one type **char** equals 8 bits’’ paradigm is entrenched in a whole generation of application software that is not going to be re-written overnight. Therefore, as X/Open is concerned with application portability, it is necessary to view the discussion of UCS support in terms of:

- How can existing non-UCS systems coexist with the new generation of UCS systems?
- What are the implications of UCS codeset support for existing XPG4 APIs?
- What implementation issues are raised by UCS support that fall outside the scope of formal standards?

These and related issues are discussed in the following chapters.

Chapter 4

# Implications for X/Open Specifications

The definition of a UCS system, as given in Section 1.2 on page 2, implies the use of UCS as the internal codeset of a system, much as ASCII-based codesets are used today. Thus, system object names would be UCS encoded (files, directories, symbolic links, special files, and so on), as would text data files. To meet these requirements within the scope of current XPG4 APIs, would require that the **char** type were defined to be large enough to hold a UCS encoded character value.

There is nothing in the current XPG4 base definition to prevent an implementor doing this. However, the more likely requirement in the short term is that suppliers will require to co-exist their existing codeset usage with support for UCS as a means of encoding text data. This has the advantage of maintaining faith with existing applications and users, while providing a migration path to UCS systems of the future.

The question must therefore be asked as to whether existing APIs are rich enough to support UCS codeset operation in such a way that existing codeset usage is not invalidated. This chapter considers the implications of this question, in terms of XPG4 Base components.

## 4.1 Internationalised System Calls and Libraries

If the tenet is accepted that the definition of the **char** type should remain as it is (that is, as mapping to an 8-bit byte), then how can UCS encoded values be accommodated?

ISO C, and hence XPG4, contain two further definitions that are relevant to this discussion:

- *multi-byte character*, which is defined as a sequence of one or more bytes (of type **char**) representing a member of the extended character set. Multi-byte character sequences can be used to represent extended characters on external storage but are generally impractical as process codes.
- *wide character*, which is a value of type **wchar_t** used to hold the wide character code, as an integral type, corresponding to a multi-byte character of the extended character set. All wide-character codes in a given process consist of an equal number of bits.

Wide-character codes thus provide a uniform size for manipulating single-byte or multi-byte text data. A wide-character code having all bits zero is the null wide-character code and terminates wide-character strings. Further, XPG4 defines that the wide-character value for each member of the Portable Character Set equals its value when used as the lone character in an integer character constant. Wide-character codes for other characters are locale-dependent and implementation-dependent.

The maximum size of a multi-byte character for a given implementation is given by the constant MB_LEN_MAX (in bytes); the size in a given locale is given by MB_CUR_MAX, which returns a value less than or equal to MB_LEN_MAX. MB_CUR_MAX is defined as a macro so that an implementation can return different values in different locales.

The ISO C language further defines syntax for expressing wide-character constant values (for example, **L'a'**) and wide-character string values (for example, **L"abcd"**). Functions are also provided for converting between multi-byte character and wide-character sequences.

The basic model for use of these types is that text sequences are represented externally as multi-byte character sequences and internally for local process manipulation as wide-character codes. Thus externally the ''one byte equals one type **char** equals 8 bits'' paradigm is maintained, while internally an application can manipulate extended codes efficiently as integral values of type **wchar_t**.

### 4.1.1 Worldwide Portability Interfaces

ISO C defines a minimal set of interfaces for manipulating wide-character codes, basically allowing conversion to and from multi-byte character sequences. XPG4 extends this set of interfaces to allow wide characters to be used wherever single-byte characters can be used in the standard C model. The additional interfaces are known as Worldwide Portability Interfaces (WPI) Thus there are functions for wide-character string manipulation, collation, conversion, printing, scanning, I/O operations, and so on. Also, whenever wide-character codes are transferred to and from external storage (disc, display devices, and so on), they are automatically converted to and from multi-byte character sequences (if applicable).

There are many advantages to this model, but the main points of interest here are as follows:

- **Efficiency**
  Internally, UCS or multi-byte codes can be manipulated as integral values of type **wchar_t**, with direct access to data arrays, tables, or whatever (indeed, internally they are UCS codes).

- **Compatibility**
  By using multi-byte character sequences as the external representation, compatibility is maintained with existing 8-bit system operation. Thus, multi-byte codes will not break file-system naming conventions, they can be processed as **char** streams by applications that are not sensitive to encodings, and can generally be used wherever single-byte character codes are currently used.
- **Interworking**
  Multi-byte character sequences do not invalidate data interchange models like XDR, and so on.

For this model to be tenable in the context of UCS, wide-character codes probably need to be maintained externally using FSS-UTF encodings. The WPI interfaces can then be responsible for conversion to and from UCS codes as text data is transferred from and to external storage. Use of FSS-UTF coded character values in a text stream also maintains ISO C text stream semantics should such a stream be processed by non-WPI functions.

The question to consider now is:

> Are the existing wide-character interfaces sufficiently complete to cater for UCS operation as defined in ISO / IEC 10646 and Unicode?

The following wide-character interfaces are defined in XPG4:

| **Worldwide Portability Interfaces** | |
|---|---|
| *fgetwc*( ) | *wcscat*( ) |
| *fgetws*( ) | *wcschr*( ) |
| *fputwc*( ) | *wcscmp*( ) |
| *fputws*( ) | *wcscoll*( ) |
| *getwc*( ) | *wcscpy*( ) |
| *getwchar*( ) | *wcscspn*( ) |
| *iswalnum*( ) | *wcsftime*( ) |
| *iswalpha*( ) | *wcslen*( ) |
| *iswcntrl*( ) | *wcsncat*( ) |
| *iswctype*( ) | *wcsncmp*( ) |
| *iswdigit*( ) | *wcsncpy*( ) |
| *iswgraph*( ) | *wcspbrk*( ) |
| *iswlower*( ) | *wcsrchr*( ) |
| *iswprint*( ) | *wcsspn*( ) |
| *iswpunct*( ) | *wcstod*( ) |
| *iswspace*( ) | *wcstok*( ) |
| *iswupper*( ) | *wcstol*( ) |
| *iswxdigit*( ) | *wcstombs*( ) |
| *mblen*( ) | *wcstoul*( ) |
| *mbstowcs*( ) | *wcswcs*( ) |
| *mbtowc*( ) | *wcswidth*( ) |
| *putwc*( ) | *wcsxfrm*( ) |
| *putwchar*( ) | *wcsxfrm*( ) |
| *towlower*( ) | *wctomb*( ) |
| *towupper*( ) | *wctype*( ) |
| *ungetwc*( ) | *wcwidth*( ) |

In addition, functions like *printf*( ) and *scanf*( ) have been extended to handle wide-characters and wide-character strings in their argument lists.

### 4.1.2 Combining Characters

The XPG4 wide-character model assumes that each code element encodes exactly one composite sequence. No facilities are provided for detecting or processing composite sequences made up of a base character (a non-combining character) and one or more combining characters. Combining characters are also referred to as floating diacritics.

Unicode has the following to say about combining characters and composite sequences:

- Combining characters are treated as separate (non-spacing) characters. When rendered, these characters are intended to be positioned relative to the preceding base character in some manner, and not to occupy a spacing position by themselves. Their input order is *after* the base character and from the centre of the base character outward.
- Composite sequences and digraphs are specified as sequences of Unicode characters; characters with one or more diacritics are thus included at no cost in code space.

The WPI model as defined in XPG4 contains no interfaces suitable for the processing of composite sequences. The following interfaces have been proposed to help alleviate this malady:

- *wcstxcpy*( ), which copies a number of composite character sequences from one wide-character string to another.
- *wcstxcat*( ), which appends a number of composite character sequences to a wide-character string.
- *wcstxcnt*( ), which determines the number of code elements in a composite character sequence.
- *wcstxwidth*( ), which determines the display width of a composite character sequence.
- *wcstxnext*( ), which advances a pointer to a wide-character string to the next composite character sequence.
- *wcstxnorm*( ), which performs a specified normalisation (conversion to lower-case, upper-case, and so on) on an input composite character sequence (**wchar_t***) and stores the result in another wide-character string.
- *wcstxattr*( ), which retrieves the attributes associated with each code element of a wide-character input string.

These functions, or something equivalent, will need to be added to the XPG4 API to facilitate the processing of composite sequences and floating diacritics, which is required for ISO/IEC 10646 level 2 and level 3 implementation.

### 4.1.3 Portable Character Sets

XPG4 defines the following character sets, which must be present in all locales, for the porting of text data between XPG4-compliant implementations:

- The portable filename character set. This defines the ISO/IEC 646 alphanumeric characters, plus period (.), underscore (_) and hyphen (-).
- The portable character set. This defines the same characters as the portable filename character set, plus the special characters in the ASCII range 0x21 to 0x40, and the control characters alert, backspace, tab, newline, vertical-tab, form-feed, carriage-return, space and the null character (NUL).

As their names imply, the portable filename character set is used for naming file objects and the portable character set is used for the encoding of text files. Use of these character sets is only significant when file objects or text files are moved between X/Open systems.

XPG4 does not specify an encoding scheme for these character sets. However, by common practise, and because no data announcement mechanism is currently defined between systems, they are usually encoded using ISO/IEC 646 (ASCII) 7-bit codes.

**Note:** The definition of the POSIX locale in XPG4 defines character classification and collation order according to ASCII codeset rules, which also includes all the control characters in the ASCII range 0x00 to 0x1f. However, the actual encoding scheme for the POSIX locale is not defined and no default for **<code-set-name>** is given.

In terms of the portable character sets described above, there is clearly no problem with either ISO/IEC 10646 or Unicode, as they both include all the desired characters. The POSIX locale also defines the ASCII control characters, which again is satisfactory in terms of the characters it defines. Encoding is another matter.

Within the limitations of current XPG announcement mechanisms, use of 16-bit or 32-bit encodings would cause problems when using the portable filename character set or the portable character set to transfer filenames or text files between UCS and non-UCS systems. A non-UCS system has no way of determining the encoding scheme of data it is importing, and the presence of nulls and slashes in the upper half of UCS-2 characters (for example) would violate local file-system and text-file semantics.

The answer, again, is to use FSS-UTF for text interchange between systems. FSS-UTF preserves 7-bit encodings in a single-byte, so ASCII and FSS-UTF codings are indistinguishable when encoding 7-bit ASCII data. If 8-bit or larger encodings were present, they might not make sense on a non-UCS system but they would not break it.

## 4.2 Commands and Utilities

The XPG4 commands (ISO POSIX-2 compatibility set) are defined to work correctly in an international environment; that is, their behaviour is localised according to the language, territory and codeset announced in the environment of the process in which they are executed. Thus they should operate correctly when called from either the shell or from an application (by means of *system*( ), *popen*( ), and so on).

The advent of UCS places a further requirement on the operation of system commands and utilities, in that to cater for wide-character codes they should now process text data by means of the WPI interfaces (possibly augmented with the extra interfaces described in the previous section).

This is a slightly tenuous statement of requirements, as XPG4 defines that commands shall work correctly in an international environment, but it does not define support for any specific codesets. It could be argued that the above requirement is already present in XPG4; that is, to support multi-byte codeset operation correctly. However, there is no verification of this at present, as multi-byte codeset support is itself optional.

Indeed maybe this is a prudent point at which to review XPG4 codeset recommendations and requirements — basically, there are none, except in the context of specific commands and utilities. XPG3 recommended use of ISO 8859-1 for text interchange between systems located in West Europe and North America, the intention being to extend these recommendations to cover other geographic areas when APIs were updated to cater for the requirements of more complex languages. In the event, no agreement could be reached, other than interchange based on the portable character set (which does not define encodings).

Maybe ISO/IEC 10646 and Unicode provide a solution to this problem, given that the eventual goal is to include all the characters of all the languages in the world. A possibility is to specify the ISO/IEC 10646 UCS-2 character set for text interchange between X/Open systems, but encoded using FSS-UTF so that text can be processed on both UCS and non-UCS systems.

The remainder of this chapter considers implications of UCS for the operation of specific commands and utilities defined in XPG4.

### 4.2.1 Codeset Conversion

XPG4 includes the *iconv* family of functions and a command for converting the encoding of characters from one codeset to another. Character encodings in either codeset may include single-byte or multi-byte values. The results of specifying invalid characters in the input stream are undefined, onus being placed on system documentation to describe implementation behaviour in such circumstances.

If FSS-UTF were adopted as a standard code for exchanging text data between X/Open systems, then the description of the *iconv* command and functions would have to be updated to reflect this requirement; that is, all conforming systems would be required to provide conversion facilities between their local process or file codes and FSS-UTF.

The thorny question of what to do if imported text contains FSS-UTF codes not representable in the target codeset still remains. This problem already exists when presenting multi-byte values to systems that only support single-byte codesets, so it is not a new problem created by UCS encodings.

### 4.2.2 Electronic Mail

The description of the *mailx* command in XPG4 contains the following warning in its **APPLICATION USAGE** section:

> The *mailx* utility cannot guarantee support for all character encodings in all circumstances. For example, inter-system mail may be restricted to 7-bit data by the underlying network, 8-bit data need not be portable to non-internationalised systems, and so forth. Under these circumstances, it is recommended that only characters defined in the ISO/IEC 646:1991 standard International Reference Version (equivalent to ASCII) 7-bit range of characters be used.

UNIX mail systems are typically based on RFC 822. In this context, messages are viewed as having an envelope and contents. The envelope contains whatever information is needed to accomplish transmission and delivery; the contents compose the object to be delivered to the recipient. RFC 822 further defines that:

- Field-names (**To:**, **From:**, and so on) must be composed of printable ASCII characters (characters that have values between 33 and 126 decimal, except colon).
- Field-bodies can be composed of any ASCII characters, except CR and LF.
- Message bodies are simply a sequence of lines containing ASCII characters, separated from the headers by a null line.

Neither the ISO POSIX-2 standard nor XPG4 define that mail must be implemented using RFC 822, but usage thereof is commonplace due to its prevalence in UNIX mail networks. Hence the XPG4 warnings above; that is, for maximum portability use only ISO/IEC 646 IRV characters and encodings (7-bit ASCII) in mail messages.

This is not an issue raised by ISO/IEC 10646 or Unicode; problems can arise when attempting to transmit ISO 8859-1 8-bit values across certain types of network. FSS-UTF does not help in this case either, as this may also contain 8-bit encodings.

The practical way around these difficulties is to use only ASCII characters in mail headers, and to encode messages by means of *uuencode* if it is required to transmit text containing 8-bit values. This is ugly but it works.

The reason for raising these difficulties here is to illustrate another area where existing systems are dependent, in practical terms, on ASCII single-octet encodings. This may be an issue for non-UCS systems attempting to transmit messages using FSS-UTF encodings, and it almost certainly has implications for UCS systems wanting to join UNIX mail networks.

A secondary issue in this area is concerned with inter-system mail between UCS and non-UCS systems, even where the above restrictions do not apply. In this case, it would again be sensible to recommend FSS-UTF encodings for message headers and bodies, and further restrict character usage in headers to only those characters defined in the portable character set. This latter restriction is necessary to prevent invalidation of existing machine and user naming conventions.

### 4.2.3 Portable Archives

The portable archive utility *pax* reads, writes or lists the contents of portable archives. The format of a portable archive may take one of several forms, that is:

- The extended *cpio* interchange format.
- The extended *tar* interchange format.

These formats are specified in Section 10 of the ISO POSIX-1 standard (ISO/IEC 9945-1) and in the Commands and Utilities volume of XPG4. The only difference between the two is that the ISO POSIX-1 standard describes archive header fields in terms of bytes; XPG4 is slightly more specific and describes these fields in terms of octets. For example, ISO POSIX-1 contains the following description of a *tar* header:

| Field Name | Byte Offset | Length |
|---|---|---|
| **name** | 0 | 100 |
| **mode** | 100 | 8 |
| **uid** | 108 | 8 |
| **gid** | 116 | 8 |
| **size** | 124 | 12 |
| **mtime** | 136 | 12 |
| **cksum** | 148 | 8 |
| **typeflag** | 156 | 1 |
| **linkname** | 157 | 100 |
| **magic** | 257 | 6 |
| **version** | 263 | 2 |
| **uname** | 265 | 32 |
| **gname** | 297 | 32 |
| **devmajor** | 329 | 8 |
| **devminor** | 337 | 8 |
| **prefix** | 345 | 155 |

ISO POSIX-1 further goes on to state the following about encodings within the header block (assuming **magic** is set to "ustar"):

- All characters are represented in the coded character set of ISO/IEC 646.
- Numeric fields are represented as strings containing octal numbers using digits from ISO/IEC 646 IRV.
- The **uname** and **gname** fields contain the ISO/IEC 646 IRV representation of the owner and group of the associated file respectively.

The description of the extended *cpio* format contains similar wording about *cpio* header blocks.

Clearly it is necessary to fix encodings for a portable archive format, and in the XPG4/POSIX case the chosen codeset is ISO/IEC 646. An XPG4-compliant or POSIX-compliant UCS implementation is therefore required to support ISO/IEC 646 codeset operation, at least for the purposes of reading or writing portable archives.

### 4.2.4 The uu* Utilities

XPG4 contains the same warnings in sections describing the *uucp* family of utilities that it does for *mailx*; that is, networks may be restricted to 7-bit data, 8-bit data and filenames may not be portable to non-internationalised systems, and so on. However, in the case of *uucp*, the problem is more severe.

Typical implementations of *uucp* use a series of control files to control the configuration of a *uucp* network. Taking the SVR4 implementation of *uucp* as a model, several such files are defined, examples of which are as follows:

- The **Config** file, which allows an administrator to over-ride certain *uucp* operational defaults (for example, protocol usage and packet size).
- The **Devices** file, which contains information about all devices that may be used to establish a link to a remote system.
- The **Systems** file, which contains information needed to establish connection to a remote system, including scripts to control the login dialogue.
- The **Permissions** file, which gives the permissions that remote systems have with respect to login, file access and command execution.

The important point here is that all these files contain text scripts of one form or another, used to drive the *uucp* system when communicating with remote systems. Any text that is used locally, such as that contained in the **Permissions** file, is only of interest to the local *uucp* implementation and can be encoded in the default local codeset. The **Systems** file, however, contains dialogues used for logging in to remote systems, in the form:

```
expect[-send-expect]...
```

where both the expect and send sequences contain the actual encodings of text that flows between the two systems, including escape sequences such as NUL, backspace, carriage return, newline, and so on. By practical implication, if not by specification, the *uucp* system is inextricably tied to 8-bit encodings and ASCII.

In many ways this is similar to the discussion about portable archives, except in this case the standards do not mandate any specific codeset usage. However, it is self-evident that for inter-system operations of this kind, either there must be some common understanding of data mappings and character encodings, or some form of data transformation layer must exist within the protocol stack.

The situation with regards to remote command execution by means of *uux* is even more problematic. For example, consider the command:

```
uux "!diff a!/usr/file1 b!/usr/file2 >!file.diff"
```

This command collects **file1** from system **a**, **file2** from system **b**, executes *diff* on the local system and writes the results to file **file.diff** in the current directory.

There is no data announcement present in this mechanism, so the assumption is that codeset usage for filenames and the encoding of text files is the same on all systems in a *uucp* network (for which *uucp* permissions permit a command of the above form). This works today because most systems work in ASCII, or an 8-bit or larger codeset that supports ASCII encodings within it (for example, ISO 8859-1).

This model would be broken if one of the above systems were a UCS system using ISO/IEC 10646 or Unicode multi-octet encodings in either filenames or text files. It would not necessarily be broken by an XPG4-compliant system using multi-byte encodings.

The *uucp* description in XPG4 is slightly remiss at the moment, in that it does not define encoding requirements for *uucp* and *uux* interworking (for example, like those defined for *pax*). However, even if it did, it is not obvious how a UCS system could plug into such a network.

FSS-UTF does not really help here either, and it is difficult to know what to recommend, unless it is deemed realistic to propose that UCS systems are obliged to provide an ISO/IEC 646 locale for the purpose of supporting *uucp* transfers? This also affects inter-system UNIX mail (as distinct from X.400 or X.500 mail), as typically the transmission of mail messages between systems is implemented using *uucp*.

Chapter 5

# Other Considerations

The previous chapter describes ways in which it is or is not possible to deploy ISO/IEC 10646 using current interfaces. There are at least two additional ways to use the codeset that also need to be considered: that is, with codeset-specific data types and interfaces, or as a well-known process code. This chapter deals with these considerations.

## 5.1 Data Types and Interfaces Specific to ISO/IEC 10646

In XPG4, X/Open added the WPI interfaces as a way to support requests for codeset independence. Whereas the XPG3 interfaces were oriented toward single-byte codesets, XPG4 removes many such dependencies. Since codeset usage differs from country to country and user to user, codeset-independent software has been seen as a good way to provide a single system that works around the world.

While the WPI interfaces can handle many encodings, they are not completely codeset-independent (CSI). For example, truly CSI interfaces would be able to support Level 2 and 3 combining characters. The XPG4 interfaces cannot.

Although some people still stress the importance of codeset independence, there is growing sentiment for interfaces for ISO/IEC 10646 only, or codeset-dependent (CSD) interfaces. Advocates of data types and interfaces for ISO/IEC 10646 often reason that it is easier to write programs that know and take advantage of a single character encoding. They note that prior to the advent of internationalisation, most programs were written with the assumption that all data was ASCII-encoded.

ASCII-only software was unacceptable because it basically only handles English, but since ISO/IEC 10646 is a universal set, it is supposed to support all characters in all languages. Some believe it will become ‘‘the new ASCII’’ and therefore remove the need for codeset independence. Another reason for adding data types and interfaces for ISO/IEC 10646 only is that several future operating systems, including Microsoft’s NT and a future Apple Computer operating system, have such types and interfaces.

Among the proposals for CSD support are:

- That a new data type be created that can contain UCS-2, Level 3 data only.
- That two new types be created; say, **ucs2_char** (16 bits) and **ucs4_char** (32 bits). There are various opinions on the level or levels these types should support.

In addition to the data types, there also are proposals for APIs specific to ISO/IEC 10646. These would be in addition to **char** and **wchar_t** based interfaces, for example:

Existing:

```
char *strcat (char *s1, const char *s2);
wchar_t *wcscat (wchar_t *ws1, const wchar_t *ws2);
```

New (names are placeholders only):

```
ucs2_char *ucs2cat (ucs2_char *u2s1,
       const ucs2_char *u2s2);
ucs4_char *ucs4cat (ucs4_char *u4s1,
       const ucs4_char *u4s2);
```

There are two scenarios under which codeset-specific interfaces could be added:

1. If or when ISO/IEC 10646 becomes ''the new ASCII''
2. As an implementation alternative to the existing codeset-independent APIs.

If a single form of ISO/IEC 10646 becomes the new ASCII, codeset-specific data types and APIs may be widely used. XPG4's WPIs must be somewhat general in nature in order to support many different encodings. Such generality at times has an adverse impact on efficiency and performance, and means that underlying implementations are often more complex than they would be if the interfaces supported a single codeset. Dedicated ISO/IEC 10646 interfaces could be tuned to a single form or group of forms. (Note that as the number of supported forms increases, implementations become more complex and the efficiency and performance gains may decrease.)

The question is, how likely is it that a form of ISO/IEC 10646 will become the only supported encoding on all or most computer systems? Complete uniformity seems unlikely. Even when ASCII dominated the U.S. market, it was not the only supported codeset; for example, EBCDIC-based systems had a significant share of the market. Today, people around the world use many different codesets, and it is often difficult to move them from one to another.

Users resist codeset changes for numerous reasons, an important one being that the change results in a difference in the amount of space their data consumes. Data stored in national or regional codesets usually consumes less space than does the equivalent UCS or UTF encoding. Given that most users work in a single language, and that they have terabytes of existing data, they may see no advantage to giving up national sets (particularly single-byte ones) in favour of ISO/IEC 10646.

Another obstacle to ISO/IEC 10646 becoming the new ASCII is its support of multiple forms. There is only one encoding for ASCII, so it is possible to write programs that depend on that one encoding. ISO/IEC 10646, however, includes the 2-octet and 4-octet forms, three conformance levels, and allows subsetting. There are also the semi-official UTF forms. There is considerable industry support for FSS-UTF as the multi-byte form of ISO/IEC 10646, but there is no sign of consensus on the form or forms to support within a **wchar_t** or in ISO/IEC 10646-specific interfaces.

Because of legacy programs, storage size, user inertia, and other factors, it seems likely that ISO/IEC 10646 will be one of several supported codesets rather than being the only one. In addition, the trend in information technology is toward increasing interoperability and distributed computing, which implies that ISO/IEC 10646-based systems and those that support other codesets will have to interoperate. To enable interoperability between such systems, a number of issues must be resolved, based on both customer requirements and technology trends.

Despite the prediction that ISO/IEC 10646 will be one of many codesets rather than the only one, there is still an argument for adding ISO/IEC 10646-only data types or interfaces. As noted, the WPIs were not designed to handle ISO/IEC 10646's combining characters, and it is not feasible in existing implementations to use UCS forms as a multi-byte encoding. New data types or interfaces could be designed to take advantage of ISO/IEC 10646's specific attributes. They would also make it possible for those who want to write CSD programs to do so.

The advantage of CSD programs is that they can include logic that is geared toward the single supported codeset. Instead of having generalised routines that are capable of handling many codesets, such programs can streamline the logic and hard-code character-handling information.

Such programs may need a way, however, to get data into the single form they support. If an application processes a single form of UCS only, it can either assume all data is in that form, or it

needs the appropriate converter modules. If the application makes assumptions, they may be incorrect, and if it requires the use of converter modules and such modules are unavailable, it is not able to process the data. This is analogous to the current locale model; that is, data is assumed (sometimes incorrectly) to match the current locale, and a requested locale may not always be available.

A disadvantage of CSD data types and interfaces is that they cannot metamorphose over time. They therefore may not be able to keep up with changing requirements. In early 1992, if a vendor had chosen to write programs that depended on UTF-1 as the multi-byte form of ISO/IEC 10646, it would have had difficulty moving to the technically superior FSS-UTF version that was approved later in the year. Similarly, ISO/IEC 10646 had two levels in early 1992, but an extra one was added late in the year and there are proposals to add another. If CSD interfaces and data types had been created to support one of the early levels, it might be difficult or impossible to change to the other levels.

Another disadvantage of CSD types and interfaces is that they may not always support the full range of user-required characters. Although ISO/IEC 10646 has a very large repertoire, it does not currently include all the ideographs in the 1992 version of CNS 11643 (the Taiwanese codeset). If these ideographs are not added to ISO/IEC 10646, CSD interfaces would not be able to process such characters.

As requirements, level definitions, and character repertoires change over time, CSD data types and interfaces that are geared toward today's realities may become less useful.

It also is important to consider how CSD types and interfaces might fit in with existing internationalisation interfaces. Developers often find the current dual set of interfaces based on **char** and **wchar_t** confusing.

## 5.2 ISO/IEC 10646 as the Well-known Process Code

Type **wchar_t** is a semi-opaque type in that ISO C does not define its size or contents. Three sizes are common in current implementations: 8-bits, 16 bits and 32 bits.

As implemented in several operating systems, the contents of **wchar_t** differ depending on locale. Because multi-byte encodings have different characteristics, these implementations have separate algorithms for converting the multi-byte characters to **wchar_t** representations. Thus, an eucJP (Japanese EUC) **wchar_t** encoding differs from the Japanese SJIS version, and both differ from the ISO 8859-1 version.

Maintaining flexibility in the size and contents of type **wchar_t** comes with a cost. It means wide characters are not exchangeable between processes and that programs must be written such that they do not depend on or take advantage of any **wchar_t** encoding. There have been proposals to handle process code another way: to designate a single size and a single encoding for **wchar_t** so that applications can exchange process code and developers can write programs that take advantage of the single representation. The most common suggestions are that either UCS-2, Level 1, or UCS-2, Level 3 be designated as the single, well-known process code. Under this proposal, whenever converting a multi-byte character to wide character form (that is, whenever using *mbtowc*( ) or *mbstowcs*( )), the only valid wide-character encoding would be the selected UCS form.

If there is to be a well-known process code, Unicode or a form of ISO/IEC 10646 are the only viable candidates. They are the only codesets with a large enough repertoire to satisfy most users' requirements.

On an implementation level, there are both similarities and differences between the current, multiple **wchar_t** representation model, and an ISO/IEC 10646-based only model. For systems that support more than one multi-byte encoding, both models require multiple sets of converters to the wide character form. For example, if a system supports ISO 8859-1..n, Japanese eucJP and SJIS, and Taiwanese EUC, it needs the appropriate *mbtowc*( ) converters for each of these, regardless of the destination process code. Note that in a multiple **wchar_t** representation model, some codesets share the same converter, but that is not the case in an ISO/IEC 10646-only model. Consider the ISO 8859 codesets. In a multiple **wchar_t** model, all these sets usually use the same *mbtowc*( ) logic (typically, just a zero-padding of the single-byte characters out to the system's **wchar_t** size). In an ISO/IEC 10646-only model, each set needs its own converter because the ISO 8859 values do not match the ISO/IEC 10646 values (except for ISO 8859-1).

A difference between the multiple and ISO/IEC 10646-only models is that the former often relies on algorithms for conversion while the latter often needs table lookup. The latter's performance may therefore be slower than the former. Note that UTF to UCS conversions are done by means of an algorithm. Therefore, a way to remove the performance disadvantage with an ISO/IEC 10646 only model is to allow only UTF forms as multi-byte encodings.

In addition to implementation considerations around a single well-known process code, there is also the question of which one (if any) to designate as that code. As noted earlier, the XPG4 **wchar_t** interfaces do not currently support combining characters, so it is not feasible now to designate UCS-*, Level 2 or 3 as the well-known process code. UCS-*, Level 1 is less than ideal for other reasons. With combining characters, ISO/IEC 10646 has a nearly infinite repertoire of composite character sequences, but UCS-*, Level 1 necessarily has a smaller repertoire. If UCS-*, Level 1 is designated as the well-known process code, technologies will not be able to process languages that are only fully supportable in ISO/IEC 10646 through the use of combining characters, and will not be able to support characters (like the ideographs in the revised Taiwanese set) that have not been added to UCS-*.

Even if one form of ISO/IEC 10646 (or any other encoding) is ideal as a process code, there are reasons not to specify a well-known process code. Since ISO C does not define the size or contents of **wchar_t**, operating system suppliers implement it differently. While X/Open might select one size and one encoding, system suppliers are free to choose something different. This means any code that X/Open members write to take advantage of an X/Open definition may not be portable to other ISO C-compliant operating systems. Naturally, this is not a disadvantage for companies for whom application portability is not a goal.

## 5.3 Implementations of ISO/IEC 10646 or Unicode Systems

In defining a strategy for handling ISO/IEC 10646, major forces in the industry (such as Microsoft's NT) must be considered; X/Open-compliant technologies must be capable of interacting with these forces.

Microsoft provides Unicode support in its NT release. It does so using a single source, dual object model, where one object supports Unicode only (currently without combining characters) and the other handles ASCII-based encodings. Support for combining characters is planned for a future release. The NT implementation also includes the following features:

- NT uses **wchar_t**, but only allows it to be 16 bits and to contain only Unicode. NT hard-codes in dependencies on the size and contents of **wchar_t**. In essence, NT has chosen Unicode to be its single, well-known process code.
- NT uses Unicode as a multi-byte file code. As noted above, Unicode and UCS-* are not permissible as multi-byte file codes on ISO C-compliant eight-bit byte systems.
- NT uses wide character functions that have names similar to the XPG4 WPI functions, but are different because they can only process Unicode data. In some cases, the Microsoft names and syntax match the XPG4 versions; in others, they do not.

Microsoft can make some of these design decisions because application portability is not a high priority in NT. X/Open faces different constraints, but there are things X/Open can do to ease interoperability with NT or other ISO/IEC 10646-based implementations. See Chapter 7 for specific recommendations.

Chapter 6

# Implementation Issues

This chapter contains a list of implementation issues. Strictly speaking these fall outside the scope of existing XPG specifications, but they need to be considered.

## 6.1 UCS as a File Code

This discussion assumes an implementation where the developer requires to maintain compatibility with existing 8-bit codeset usage, while providing support for UCS through the WPI functions. In this scenario, UCS codes would be presented as objects of type **wchar_t**. Externally they would be represented in an implementation-defined manner.

It is this implementation-defined external encoding that causes a number of questions to be raised. For example:

- Given that no data tagging is currently defined for X/Open-compliant or POSIX-compliant systems, how does an implementation determine the encoding of a file (remembering that in this example there is a mix of external encodings)?
- What external encoding should be used to represent UCS codes?
- How does the character special file mechanism of UNIX systems lend itself to coping with multiple encodings when displaying graphic symbols or reading characters from an input device?

The data tagging dilemma is solved in part by the locale announcement mechanism defined in XPG4, which allows the user to specify localisation requirements for language, cultural conventions and codeset operation. As an example, it might be expedient to define that in a locale that defines UCS-2 (or Unicode) as the codeset, the WPI I/O interfaces assume translation from the external UCS representation (whatever that might be) to UCS-2 codes whenever data is moved between main store and external storage. Non-WPI interfaces would continue to operate on byte streams as at present.

This leaves responsibility with the application to determine whether or not it can process UCS codes (that is, by use of the WPI), though the question remains as to what happens if the WPI is used in a locale that does not define UCS as the required encoding? Should these interfaces always assume translation to and from the external UCS encoding of an implementation, or should they process external codes in a locale-sensitive manner?

This brings us to the debate concerning how UCS codes should be represented externally; that is, as true UCS codes, in FSS-UTF form, or in some other implementation-specific form?

- **As UCS Codes**
  The advantages of using UCS codes directly are obvious — no translation is required when reading or writing text, processing costs are cheaper, and so on. The major disadvantage is that such a file is no longer processable by means of the standard C library interfaces, as it breaks the rules for text file encodings (because null bytes could appear in the upper byte of a UCS code).

- **As FSS-UTF Codes**
  Using FSS-UTF as the external file code has the advantage that such a file would still be processable as a text stream, it could be passed through a pipe safely (for the same reasons), and it is in a form suitable for transfer to other UCS or non-UCS systems. One of the disadvantages is that, for processing by means of the WPI functions, character encodings would need to be translated to and from UCS-2 form as they pass between an application and the WPI.

  **Note:** FSS-UTF obeys the ISO C rules for text file encoding and as such would allow FSS-UTF encoded files to be processed by standard 8-bit commands and utilities. However, any locale-sensitive operations may yield surprising results, as (for example) FSS-UTF encoding of ISO 8859-1 8-bit characters uses multi-byte values.

- **In an Implementation-specific Form**
  This is really beyond the scope of the present report, but it is conceivable that UCS codes could be held externally in a form that corresponds to one or more of the existing multi-byte codesets. In effect, translation within the WPI would then be equivalent to using the *iconv**( ) interfaces for codeset translation.

Whichever of these is chosen, an implementation will be faced with the problem of what to do when these codes are presented to device drivers for transfer to or from peripheral devices. For example, the device driver may need to recognise whether it is handling a local 8-bit code or the external form of a UCS code.

Consider the case of a driver that supports the POSIX General Terminal Interface. If ICANON is set, then it may be requested to perform input pre-processing. Similarly if OPOST is set, it performs output post-processing. In either case, this requires that the driver recognise certain control character encodings (for example, carriage-return, newline, horizontal tab, and so on). The codes it should use for this are given in the **c_cc** structure associated with each control terminal, which are normally set when a session is created and are not locale-sensitive. With UCS encoded data, it could now be presented with different representations of these characters, possibly from within the same session.

FSS-UTF protects this mechanism to a degree, by preserving ISO/IEC 646 encodings within a single-octet encoding. However, other problems arise with FSS-UTF when standard 8-bit codes are represented in multi-octet form. In this case, it is probably reasonable for an implementation to require that such files are converted to the local system codeset before being presented to such drivers.

## 6.2 More on Combining Characters

As indicated earlier in the report, ISO / IEC 10646 identifies three levels of conformance:

- Level 1, which does not have any combining characters.
- Level 2, which allows a restricted subset of combining characters.
- Level 3, which allows a full set of combining characters.

There are some issues to be considered, as described in the following sections.

### Unlimited Combining Characters

ISO / IEC 10646 Level 2 and 3 support does not restrict the number of combining characters that can follow a base character. Thus, the length of a composite sequence is not fixed and cannot be determined until the entire sequence has been processed. This has implications for the efficiency of handling composite sequences.

### Variable Ordering of Combining Characters

ISO / IEC 10646 does not define the order of combining characters, except that all combining characters must follow a base character. When the combining characters do not interact in presentation, the resulting glyph formed from different combinations of a base character and combining characters may appear the same. For example:

<A>-<caron>-<ogonek>

is the same as

<A>-<ogonek>-<caron>

This flexibility has implications for such things as collation, which must ensure that all forms of the same collation symbol have equal weight.

### Pre-composed Characters and Combining Characters

According to the level 3 definition of combining characters, some graphic symbols can be encoded either as a single graphic character or as a composite sequence. For example:

<A-acute>

is the same as

<A>-<acute>

The first form occupies a single UCS code value, while the latter form occupies two UCS code values. Again, this has implications for character processing and comparison.

## 6.3 Locales

XPG4 defines a single standard locale, the POSIX or C locale, that must be supported on all conforming implementations. Within this definition, it describes the character set and localisation requirements for collation, classification, time and date formats, number formats, and so on. However, XPG4 and POSIX carefully avoid defining any particular codeset usage in this locale.

The POSIX locale could be implemented using a wide variety of codesets, either single-byte, multi-byte or UCS. The character set defined in this locale is currently limited to the ASCII set of graphic characters and control characters, all of which are supported by ISO/IEC 10646 (in its various forms) and Unicode.

This is the sole provision in XPG4 to the notion of a standard locale. It has been suggested that in the future there will also be a need for a default UCS locale or locales, for the following reasons:

- Many character properties are constant across languages and cultures, so duplicating them for all UCS characters in every supported locale may be prohibitively expensive.
- To eliminate duplication within a single system and to aid consistency between systems, a default UCS locale carrying common character properties should be defined.
- The existence of such a default UCS locale should enable other locales to provide only overrides.

Such a locale could be registered by means of the locale registry. Initially, there could be one such definition covering UCS-2 and Unicode. In the future, there may also be a need for a standard UCS-4 locale definition.

Another issue here is concerned with how the Han characters are displayed in locales where these characters are not part of the designated language.

ISO/IEC 10646 and Unicode define CJK unified ideographs (unified Han) character codes. Moreover, for each character code, multiple glyphs are maintained for the different languages to which these characters are relevant (simplified Chinese, traditional Chinese, Japanese and Korean). These glyphs look similar but are different.

The question that arises is, when trying to display the Han characters in a locale that does not include one of the above languages, which glyph should be used for display (given that a UCS locale includes all characters irrespective of language)?

Chapter 7

# Conclusions

The adoption of ISO/IEC 10646 and Unicode will depend on the business needs of individual suppliers. It is not the responsibility of a body such as X/Open to try and direct how quickly or with what urgency this migration will occur, or indeed that it will or should occur at all. It is, however, incumbent on such a body to ensure that adoption of these standards is not prohibited or unnecessarily hindered by conformance requirements to existing and proposed X/Open specifications.

This chapter contains a list of plans covering changes to the Base APIs for UCS support and on interworking between UCS and non-UCS systems, registration of FSS-UTF, and areas where no changes are planned. It also gives an indication of issues for verification and branding raised by UCS support.

## 7.1 Plans

### 7.1.1 Changes

The following changes will be implemented in X/Open specifications.

- FSS-UTF will be endorsed as the preferred multi-byte representation of the repertoire of characters in ISO/IEC 10646.
- Continuing support of other multi-byte encodings will be endorsed. FSS-UTF is an addition to other multi-byte encodings, not a replacement of them.
- The requirements for supporting composite sequences will be identified and a proposal will be developed to satisfy those requirements. This will make it possible to use Level 2 or 3 data in a wide-character process code.
- All systems will be required to provide conversion facilities between their local process or file codes and UCS codes, including FSS-UTF, by means of the *iconv**( ) family of functions and the *iconv* command.

This covers the changes for XPG4 Base components. Other components and profiles must be evaluated in the context of UCS support; for example, are APIs limited to 8-bit codeset operation, are there assumptions within interworking mechanisms about underlying codeset usage or data lengths, can data be passed transparently, and so on.

### 7.1.2 Registration of UTF-2

The FSS-UTF transformation form of encoding ISO/IEC 10646 UCS-2 and UCS-4 coded characters has been recommended to to SG2/WG2 for registration. It has been allocated the name UTF-8.

FSS-UTF will also be recommended to ECMA for registration with ISO 2375 in order to obtain an ISO 2022 ‘‘other coding method’’ announcement sequence.

### 7.1.3 No Change

X/Open specifications will continue to allow multiple **wchar_t** representations. That is, X/Open will not designate a single, well-known process code. This means existing implementations of XPG4 interfaces will continue to be viable, and also allows support for characters that may not exist in the chosen process code.

At present X/Open has no plans to create data types or interfaces specific to ISO/IEC 10646.

## 7.2 Verification Issues

The Base verification suite (VSX4) currently defines pseudo-locales for the testing of internationalisation functionality. No codesets are defined in XPG4, so no specific codeset verification is performed. In the context of the WPI interfaces, XPG4 defines that the size of the **wchar_t** type is implementation defined and can take any value (including single-byte). This requirement was very specific, in that a number of suppliers who were only interested in the European market did not want to be forced into providing support for multi-byte codesets. Thus, the WPI interfaces must be supported, but it is optional whether or not they work with multi-byte codesets.

It is also intended to provide verification for specific codeset operation at some stage. It would seem reasonable to extend this form of branding to cover UCS codeset usage, although it is doubtful whether this could or should be mandated in the Base.

Commands branding should be updated to verify the extra requirements placed on *iconv* above; that is, to ensure codeset conversion from and to UCS codes. This may require a more precise definition of what happens if a UCS code is not representable in the local codeset.

# Glossary

**basic multilingual plane**
In ISO / IEC 10646, Plane 00 of Group 00.

**byte**
An individually addressable unit of data storage that is equal to or larger than an octet, used to store a character or a portion of a character; see **character**. A byte is composed of a contiguous sequence of bits, the number of which is implementation-dependent. The least significant bit is called the *low-order* bit; the most significant is called the *high-order* bit. Note that this definition of *byte* deviates intentionally from the usage of *byte* in some international standards, where it is used as a synonym for *octet* (always eight bits). On a system based on the ISO POSIX-2 DIS, a byte may be larger than eight bits so that it can be an integral portion of larger data objects that are not evenly divisible by eight bits (such as a 36-bit word that contains four 9-bit bytes).

**canonical form**
In ISO 10646, the form with which characters of the coded character set are specified using four octets to represent each character.

**character**
A member of a set of elements used for the organisation, control or representation of data.

**character set**
A finite set of different characters used for the representation, organisation or control of data.

**character string**
A contiguous sequence of characters terminated by and including the first null byte.

**coded character set (codeset)**
A set of unambiguous rules that establishes a character set and the one-to-one relationship between each character of the set and its bit representation.

**combining character**
A member of an identified subset of the coded character set of ISO / IEC 10646 intended for combination with the preceding non-combining graphic character, or with a sequence of combining characters preceded by a non-combining character.

**composite sequence**
A sequence of graphic characters consisting of a non-combining character followed by one or more combining characters.

**Notes:**

1. A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence.
2. A composite sequence is not a character and therefore is not a member of the repertoire of ISO / IEC 10646.

**control character**
A character, other than a graphic character, that affects the recording, processing, transmission or interpretation of text.

**diacritic**
(1) A mark applied or attached to a symbol in order to create a new symbol that represents an entirely new value; (2) a mark applied to a symbol irrespective of whether it changes the value of that symbol. In the latter case, the diacritic usually represents an independent value (for example, an accent, tone, or some other linguistic information). Also called diacritical mark, or diacritical.

**empty string**
A string whose first byte is a null byte.

**file code**
The representation of text when it is stored on some external storage medium (for example, magnetic disk). File codes are implementation-defined.

**graphic character**
A character, other than a control character, that has a visual representation when handwritten, printed or displayed.

**internationalisation**
The provision within a computer program of the capability of making itself adaptable to the requirements of different native languages, local customs and coded character sets.

**locale**
The definition of the subset of a user's environment that depends on language and cultural conventions.

**localisation**
The process of establishing information within a computer system specific to the operation of particular languages, local customs and coded character sets.

**magic**
A field in the *tar* header block.

**non-spacing characters**
A character, such as a character representing a diacritical mark in the ISO 6937:1983 standard coded character set, which is used in combination with other characters to form composite graphic symbols.

**null byte**
A byte with all bits set to zero.

**null pointer**
The value that is obtained by converting the number 0 into a pointer; for example, (**void** *) 0. The C language guarantees that this value does not match that of any legitimate pointer, so it is used by many functions that return pointers to indicate an error.

**null string**
See **empty string**.

**octet**
An ordered sequence of eight bits considered as a unit.

**portable character set**
The collection of characters that are required to be present in all locales supported by X/Open-compliant systems:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 ! # % ^ & * ( ) _ + - = { } [ ]
: " ~ ; ’ ‘ < > ? , . | \ / @ $
```

Also included are <alert>, <backspace>, <tab>, <newline>, <vertical-tab>, <form-feed>, <carriage-return>, <space> and the null character, NUL.

This term is contrasted with the smaller *portable filename character set*.

**portable filename character set**
The set of characters from which portable filenames are constructed. For a filename to be portable across implementations conforming to the **XBD** specification and the ISO POSIX-1 standard, it must consist only of the following characters:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
a b c d e f g h i j k l m n o p q r s t u v w x y z
0 1 2 3 4 5 6 7 8 9 . _ -
```

The last three characters are the period, underscore and hyphen characters, respectively. The hyphen must not be used as the first character of a portable filename. Upper- and lower-case letters retain their unique identities between conforming implementations. In the case of a portable pathname, the slash character may also be used.

**process code**
The representation of text when it is manipulated by a program (for example, for classification, conversion, comparison, and so on). Process codes are implementation-defined.

**script**
A set of graphic characters used for the written form of one or more languages.

**string**
A contiguous sequence of bytes terminated by and including the first null byte.

**wide-character code**
An integer value corresponding to a single graphic symbol or control code.

**wide-character string**
A contiguous sequence of wide-character codes terminated by and including the first null wide-character code.

# Index