Snapshot

Distributed Internationalization Framework

THE Open GROUP

In conjunction with

UniForum

[This page intentionally left blank]

X/Open Snapshot

Distributed I18N Framework

X/Open Company Ltd.

X/Open Snapshot

Distributed I18N Framework

ISBN: 1-85912-079-2
X/Open Document Number: S503

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

Chapter 1 Introduction **1**

Chapter 2 The Global Locale Model **3**

2.1 What is a Locale? 3
2.2 The setlocale( ) Function 4
2.3 Object-oriented Software 5
2.4 Layered Software 6
2.5 Threaded Software 6
2.6 Stateful Encodings 7
2.7 Summary 7

Chapter 3 The Multi-locale Model **9**

3.1 Overview 9
3.2 Terms and Definitions 10
3.3 Program Flow 11
3.4 Multi-locale Functions 12
3.4.1 Locale Management Functions 13
3.4.2 Locale Information Functions 15
3.4.3 Classification Functions 17
3.4.4 Transliteration Functions 19
3.4.5 String Searching Functions 20
3.4.6 String Comparison Functions 21
3.4.7 Date, Monetary and Time Formatting Functions 22
3.4.8 Number Conversion Functions 23
3.4.9 Text Scanning and Parsing Functions 23
3.4.10 Formatted I/O Functions 24
3.4.11 Extended Wide-character Conversion Functions 25

Chapter 4 Advanced Text Handling **27**

4.1 Background 27
4.2 Complex Text 28
4.2.1 Codesets 28
4.2.2 Composite Sequences 29
4.3 Directional and Context-sensitive Text 31
4.4 Composite Sequence Functions 32

Chapter 5 Distributed Internationalisation **35**

5.1 The Problem 36
5.2 Locale Naming 38
5.2.1 String Network Locale Specifications 38
5.2.2 Token Network Locale Specifications 39
5.2.3 LocaleSpec Functions 39

5.2.4 NetSpec Functions 41
5.3 Locale Registration 42

Appendix A ISO MSE Changes 43
A.1 Data Types 43
A.2 Functions 44

Glossary 47

Index 51

Preface

X/Open

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

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

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

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

X/Open Technical Publications

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

There are two types of X/Open specification:

- *CAE Specifications*

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

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

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

- *Preliminary Specifications*

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

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

In addition, X/Open publishes:

- *Guides*

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

- *Technical Studies*

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

- *Snapshots*

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

Versions and Issues of Specifications

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

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

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

Corrigenda

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

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

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

```
      request corrigenda; topic index
```

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

This Document

This document is a Snapshot (see above). Its aim is to describe a framework for the development of APIs in a distributed computing environment, based on work done by the Joint X/Open-Uniforum working group during production of specifications for distributed internationalisation services and complex text handling.

Other, related aspects of program internationalisation are considered first, including multi-locale, multi-lingual, multi-threading, single-byte, multi-byte and universal codeset (UCS) working. Particular attention is paid to the requirements of rich text formats and stateful encodings.

Structure

This document is structured as follows:

- Chapter 1 is a brief introduction
- Chapter 2 discusses the global locale model
- Chapter 3 provides a rationale and overview of the multi-locale model
- Chapter 4 considers the requirements of advanced text handling in detail
- Chapter 5 examines the requirements of a distributed internationalisation environment
- Appendix A describes the proposed MSE changes to the ISO C standard.
- A glossary and index are provided.

Intended Audience

This Snapshot is aimed in general at all users and suppliers who wish to understand the issues surrounding internationalisation and how these can be solved. In particular, it addresses the specific issues associated with the increasingly important subject of *distributed* internationalisation.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font. In syntax ellipses (`...`) are used to show that additional arguments are optional.
- Variables within syntax statements are shown in *`italic fixed width font`*.

Trademarks

Referenced Documents

The following standards or draft standards are referenced in this document:

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

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

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] (identical to IEEE Std 1003.1-1990).

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

MSE
: ISO / IEC 9899: 1990 / Amendment 1: 1994, Multibyte Support Extensions (MSE) for ISO C.

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

Distributed Internationalisation Services
: X / Open Snapshot, December 1994, Distributed Internationalisation Services, Version 2 (ISBN: 1-85912-033-4, S308).

Layout Services
: X / Open Snapshot, December 1994, Portable Layout Services: Context-dependent and Directional Text (ISBN: 1-85912-075-X, S425).

FSS-UTF
: X / Open Preliminary Specification, May 1993, File System Safe UCS Transformation Format (FSS-UTF) (ISBN: 1-872630-96-0, P316).

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

Locale Registry
X/Open Electronic Publication, October 1993, Locale Registry Procedures (ISBN: 1-872630-94-4, G303).

XPG2
X/Open Portability Guide, five volumes, January 1987 (ISBN: 0-444-70179-6).

XPG3
X/Open Specification, 1988, 1989, February 1992 (ISBN: 1-872630-43-X, T921); this specification was formerly X/Open Portability Guide, seven volumes, January 1989 (ISBN: 0-13-685819-8, XO/XPG/89/000).

XPG4
X/Open Systems and Branded Products: XPG4, October 1994 (ISBN: 1-872630-52-9, X924).

Chapter 1

Introduction

Internationalisation facilities first appeared in Issue 2 of the X/Open Portability Guide (XPG2). This was published ahead of similar facilities defined in the ANSI C standard, and contained a rudimentary set of interfaces necessary for the development of internationalised applications; that is, applications that were free of specific codeset, language or cultural operation.

The initial XPG2 definition was derived from the Native Language Support System developed by the Hewlett-Packard Company of Palo Alto, California, enhanced by X/Open and modified in certain areas to converge with the proposed ANSI C standard. Specifically, XPG2 provided a definition of:

- transparent single-byte codeset operation
- internationalised versions of standard C library functions
- new functions that enabled applications to determine the format and setting of cultural data items
- message catalogues
- an announcement mechanism.

Most significantly, this and the proposed ANSI C standard first introduced the concept of a *locale*: an opaque object that contained all the information necessary to achieve program localisation. Also, the model that one locale equals one language, territory and codeset combination became entrenched.

XPG3 followed in December 1988. This issue of the Portability Guide was fully aligned with the POSIX.1 and ANSI C standards, except in the area of multi-byte codeset operation and the *localeconv*( ) function. The focus of XPG3 was to satisfy internationalisation requirements in Europe and North America, hence single-byte codeset operation still predominated. XPG3 branding also mandated that internationalisation support was to be included in all conforming systems.

New features introduced in XPG3 included:

- an improved announcement mechanism
- internationalised regular expressions
- an optional internationalised utility environment
- a new utility for codeset conversion (*iconv*).

The fundamental locale model remained unchanged. Initialisation of the locale was linked to the environment, and its scope was widened, but the global nature of the locale object became ever more cast in tablets of stone.

The parochial nature of XPG internationalisation was abandoned in XPG4 (July 1992), which sought to address the requirements of more complex languages, such as those manifest in Asia and the Far East. Specifically, XPG4 included:

- the Worldwide Portability Interfaces, which enabled applications to work with either single-byte or multi-byte codesets
- extended support for the Internationalised Utility Environment

- additional interfaces for date and time conversion (*strptime*( )), monetary value conversion (*strfmon*( )), and codeset conversion (*iconv_open*( ), *iconv*( ) and *iconv_close*( ))
- full conformance to the ANSI / ISO C standard
- the *localedef* and *locale* utilities.

For applications that were happy to work with one language at a time, and were only concerned with simple terminal input and output, this model was and still is adequate. However, other groups were now considering internationalisation within more complex programming paradigms, for example:

- object-oriented software
- layered software
- distributed software
- threaded software
- multi-user software
- advanced text handling software.

It became obvious immediately that the global locale model published in XPG4 was too limiting to satisfy the requirements of these groups. Some required to manipulate multiple locales simultaneously, some required more precise management over such things as stateful encodings, context sensitive or bidirectional text, while others required a model that supported distributed internationalisation.

This then was the basic set of problems that the X / Open-Uniforum Joint Internationalisation Group (JIG) set out to address in the Distributed Internationalisation Services Specification (**Distributed Internationalisation Services** snapshot), in the Layout Services Specification (**Layout Services** snapshot), and **Locale Registry** documents. These documents are specifications of functional interfaces and procedures for meeting the above requirements. This document is a framework that provides a context for those specifications and hopefully provides some insight into how and why the specifications emerged the way they have.

Chapter 2 revisits the global locale model and examines both its strengths and weaknesses, drawing towards a set of requirements for a distributed, multi-locale model of a type needed to meet the above objectives. Chapter 3 provides a rationale and overview of the multi-locale model presented in the **Distributed Internationalisation Services** snapshot, complete with a top-level introduction to its many functional interfaces. Chapter 4 considers the requirements of advanced text handling in detail, including state handling requirements in a multi-locale environment, context sensitive and bidirectional text handling. Finally, Chapter 5 examines the requirements of a distributed internationalisation environment, including distributed locale announcement, common naming procedures and locale registration.

Chapter 2

The Global Locale Model

This chapter revisits the global locale model and examines both its strengths and weaknesses, drawing towards a set of requirements for a distributed, multi-locale model.

2.1 What is a Locale?

XPG4 defines a locale as ‘‘the definition of the subset of the user’s environment that depends on language and cultural conventions. It is made up from one or more categories. Each category is identified by its name and controls specific aspects of the behaviour of the components of the system.’’

In less formal terms, a locale is a repository for control information that directs the operation of certain functions that are sensitive to the processing and presentation of data. For example, the sorting and classification of text data depends on:

- the encoding of the data
- the language of the current user.

Similarly, functions that process or present data containing such things as date and time strings need to understand prevailing cultural conventions to produce the correct results.

How this data is processed in a locale-specific way is transparent to the user and to the programmer. In the latter case, to sort strings correctly, a program simply initialises the locale with the required localisation data and then calls one of the string collation functions defined to perform locale-sensitive ordering (for example, *strcoll*( ), *wcscoll*( ), ...). The user merely has to announce which localisation data they wish to be associated with the locale.

This model is fully described in the **Internationalisation Guide**. Suffice it to state here that both the announcement mechanism and the locale initialisation procedures defined in XPG4 perpetuate the paradigm that one locale equals one language, territory and codeset combination. This is referred to as the *global* locale model, which in structural terms permits that each process environment will contain one and only one locale object.

2.2 The setlocale( ) Function

Locales are created by means of the *localedef* utility, which converts source definitions for locale categories into a localised form suitable for loading into a program locale at run time. The program locale itself is loaded by calling the *setlocale*( ) function, which in formal language terms is defined as:

```
char *setlocale(int category, const char *locale);
```

The value of *category* names all or part of the program locale and may be one of LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC or LC_TIME. LC_ALL names the program's entire locale, other settings name a specific locale category.

The *locale* argument is a pointer to a character string naming the required setting of *category*. This is the name of a locale created by the *localedef* utility. A null pointer directs *setlocale*( ) to query the current locale and return its name. A pointer to a null string ("") indicates that the locale should be initialised from the corresponding value of the associated environment variables. Thus:

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

tells *setlocale*( ) to initialise the entire program locale from settings of the environment variables LC_ALL, LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC, LC_TIME and LANG. If LC_ALL is set in the environment, this takes precedence and no other environment variable settings are used. If LC_ALL is not set, the category-specific variables are used, with LANG providing a default setting for unset variables. If LANG is not set, an implementation specific default is used instead.

This mechanism provides considerable flexibility in the setting, querying and general manipulation of a program's locale. Typically, an application will start by initialising the locale from the environment, as this provides a link between the user and the program, and then manipulate it either in its entirety or at the category level as best befits the applications needs. For example, a menu-driven or windowing application may provide an option that allows the user to change his or her codeset, language or cultural requirements dynamically.

There is also limited multi-locale support inherent in this model. For example:

```
/*
 *    Sample code showing how a locale may be saved,
 *    modified and subsequently restored.  In a real
 *    program, all return values should be checked
 *    for success, but for the sake of brevity this
 *    is not done here.
 */

char   *OldLocale, *SavedLocale;
...
OldLocale = setlocale(LC_ALL, NULL);
SavedLocale = malloc(strlen (OldLocale) + 1);
strcpy(SavedLocale, OldLocale);
setlocale(LC_ALL, "");
...
setlocale(LC_ALL, SavedLocale);
...
```

The first call to *setlocale*( ) in this example queries its current setting. As this name may be overwritten by a subsequent call, it must be saved elsewhere if it is to be reused at a later point in the program, hence calls to *malloc*( ) to allocate space to hold the string, *strlen*( ) to determine its length, and *strcpy*( ) to copy it locally. The next call to *setlocale*( ) initialises the program locale from the environment. The original locale can be restored by calling *setlocale*( ) with *locale* set to the original locale name, as indicated.

This mechanism is adequate for one-layered, single-threaded applications that have complete control over their environment and the order in which locale-specific actions are performed. However, the above example is awkward in appearance and may be expensive if, for example, the locale needs to be reloaded from filestore each time *setlocale*( ) is called.

2.3 Object-oriented Software

The locale object managed by *setlocale*( ) is a global object in every sense; that is, there is only one per program instance (process), that operates independently of program block structure. This is the antithesis of object-oriented programming, where it is far more natural to want to be able to associate a locale object with one or more other program objects.

In C++, for example, the locale object might emerge as a base class that can be inherited by other classes, which would allow an implementation to associate locale-specific behaviour with other standard classes. However, such a mechanism would require that a locale object is a visible program object in its own right, of which there may be many in existence at one time, each describing potentially diverse localisations. It further requires that calls to locale sensitive functions explicitly identify which locale object to use.

Clearly, neither of these requirements are met by the global locale model.

2.4 Layered Software

Problems associated with layered software are not dissimilar to those described above, with the added complication that each software layer requiring to manipulate the global locale object must protect all possible entry and exit points to and from the layer.

Consider the case of a windowing layer that supports different language, territory and codeset combinations in each of its windows. To manage this mechanism, the software will need to re-initialise the global locale each time it is called, with all the attendant performance overheads implied above. It must also ensure that on return to the caller, the global locale is reset to its calling state.

Thus entry points must save the calling locale, and exit points must reinstate it. This is sufficient overhead in itself to make using the global locale for this purpose an unattractive proposition, but the software layer must also protect against asynchronous returns to the calling layer, by means of signals for example. The layer is therefore faced with either masking all signals or saving all extant signal dispositions on entry, installing its own signal catching functions, and restoring the original dispositions on return to the caller. Should a signal occur while the layer is active, the software must either hold the signal pending or reset the locale and signal dispositions to their original values and manually raise the signal again.

While it is just conceivable that such an implementation could be provided using the global locale model, it would be inefficient in the extreme, error prone and difficult to manage. Again, having discrete locale objects that could be managed directly by the application provides a far more attractive and workable alternative.

2.5 Threaded Software

The requirements of multi-threaded software bring yet another set of problems. In this model multiple threads of execution can proceed at the same time within a single process, which is clearly untenable in the global locale model if different threads have conflicting localisation requirements. The model is also invalidated if the global local object is used to hold state information of any kind.

While the requirements of object-oriented software and layered software could theoretically be catered for in the global locale model, as unattractive and costly as that might be, the requirements of threaded software seem to fundamentally undermine a global solution of any kind. In this model, each thread **must** be able to manage its own locale, without interference from or reference to co-existing program threads.

2.6 Stateful Encodings

Problems associated with stateful encodings are as much a weakness in the XPG4 API as in the global locale model itself. Nevertheless they must be considered at the same time as the specific global locale problems described above.

As an example, the *mbtowc*( ) function converts a multi-byte character sequence to a wide-character code. Ostensibly, this interface purports to cope with stateful encodings by retaining state information between successive calls, within the limitation that any changes to the LC_CTYPE category of the global locale causes the shift state of the function to be indeterminate. Thus:

```
int mbtowc(wchar_t *pwc, const char *s, size_t n);
```

For a state-dependent encoding, this function is placed into its initial shift state by a call with *s* set to the null pointer. Subsequent calls with *s* set to other than the null pointer, cause the internal state of the function to be altered according to the prevailing shift state of the multi-byte stream pointed to by *s*.

Neither state introducers nor the current shift state are visible to applications that use this interface, and any change to LC_CTYPE may invalidate existing state information. Thus, were a software layer, thread or object class to change the global locale in a manner indicated in previous sections, then it could invalidate any retained state information associated with that locale, which in turn could precipitate incorrect program behaviour in the calling layer.

State information is also managed implicitly by other interfaces; for example, the stream I/O functions. Within a stateful text stream, character context is determined from the previous state introducer, or possibly introducers. In the XPG4 API, this is managed by the stream I/O functions transparently, without any possible influence by or interaction with the calling application.

And so on... The important point here is that state information associated with the global locale object complicates the situation further when considering the requirements of object-oriented, layered and multi-threaded software. In a better world, both the locale object itself and retained state information would be explicit, visible objects that an application could manage itself.

2.7 Summary

The global locale model presented in XPG4 is adequate to meet the needs of function-oriented, single-layer, single-threaded applications that are common today and will remain significant for the foreseeable future. The integrity of this model must therefore be preserved to facilitate the continued development, support and maintenance of these applications.

A new model is required to meet the needs of more complex object-oriented, multi-layered or multi-threaded software, which requires locale objects to be visible entities that can be managed by the software directly. State information should be divorced from such a locale object and presented as a separate object in its own right. All APIs that are sensitive to locale-based operations should be updated to accept a locale object as an argument, rather than using the global locale implicitly, as well as managing state objects where appropriate.

This then sets down the basic requirements for a multi-locale model of a type described in the next chapter, and in an informal way documents the rationale by which the X/Open-Uniforum Joint Internationalisation Group arrived at its requirements for producing the **Distributed Internationalisation Services** snapshot.

Chapter 3

The Multi-locale Model

This chapter provides a rationale and overview of the multi-locale model presented in the **Distributed Internationalisation Services** snapshot, complete with a top-level introduction to its many functional interfaces.

3.1 Overview

The multi-locale model presented in the **Distributed Internationalisation Services** snapshot addresses all the requirements identified in the previous section. In particular, the associated API defines a visible object to hold localisation data. This can be created, initialised and destroyed by the program, and passed as an argument to the multi-locale functions.

Note that the global locale and functions that work on this entity still exist. The multi-locale functions coexist with and complement this model by providing additional locale handling capabilities more suited to the needs of complex applications that require to handle multiple languages simultaneously, or are multi-layered or multi-threaded.

The locale object defined in the multi-locale model is in fact wrapped within another program object known as an Attribute Object. This is an implementation-defined type, other than an array type, that can hold locale specific information for the various defined locale categories. This abstraction is introduced to allow other types of information to be associated with layer or thread operation in the future (for example, security information), without necessitating the introduction of yet another program object.

The multi-locale model also defines a state object, of type **mbstate_t**, which is used to retain state information between successive calls of state sensitive interface functions. Again, this object is implementation-defined and can be any data type other than an array type. For example, this is used when converting between a coded character string and a wide-character string.

One further type is introduced, **wctrans_t**, which is a scalar type that can hold values representing locale-specific transliteration mappings. The transliteration functions are described later in this section.

3.2 Terms and Definitions

A complete glossary of terms is provided before the index. However, a number of terms new to X/Open specifications are introduced in the **Distributed Internationalisation Services** snapshot and it is worth providing a brief introduction here. This is in addition to terms like *global locale* and *locale object*, which have already been described in earlier sections. Terms specific to distributed internationalisation are defined in Chapter 5.

coded character
A code value encoded as one or more objects of type **char** that corresponds to a member of the codeset of the locale.

null coded character
A coded character with code value zero.

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

code element
Refers to a character encoded as either a wide-character code (of type **wchar_t**) or a *coded character* (of type **char***).

code element string
A contiguous sequence of *code element*s all having the same type and terminated by and including the first null *code element*. A pointer to a *code element string* is a pointer to its initial (lowest addressed) *code element*. The length of a *code element string* is the number of *code element* objects preceding the null *code element*.

3.3 Program Flow

The previous chapter included sample code that showed how to change the contents of the global locale; that is, by saving the current contents of the locale, initialising a new state, and restoring the original contents at a later time. In the multi-locale model, this sequence of events is replaced by:

```
/*
 *  Sequence of calls indicating how to create,
 *  initialise and destroy an attribute object.
 */

#include <mlocale.h>
...
    AttrObject locale;
    char *str;

    locale = m_createattrobj ();
    str = m_setlocale (&locale, LC_ALL, "");
    ...
    m_destroyattrobj (locale);
...
```

The **<mlocale.h>** header provides a definition of an **AttrObject**, and prototypes for all the multi-locale functions.

An attribute object is created by calling the *m_createattrobj*( ) function. Many such objects may be in existence at the same time without interfering with one another. The locale object within an attribute object is initialised by calling the *m_setlocale*( ) function, which is similar to *setlocale*( ) except that it takes a pointer to an **AttrObject** as its first argument.

From this example, it is readily apparent how an application can manage multiple locales in parallel; that is, by creating and manipulating multiple concurrent attribute objects. The multiple locale functions mostly accept an attribute object as their first argument, and localise their behaviour according to the settings of locale categories defined therein. For example:

```
m_wcscoll(locale, ws1, ws2);
```

will compare the contents of the wide-character strings pointed to by *ws1* and *ws2*, according to ordering rules defined in the LC_COLLATE category of the locale object associated with the attribute object identified by *locale*. This paradigm holds true for most of the *m_**( ) functions, which in the main parallel their global locale counterparts (however, see later in this chapter for further details).

As already stated, concurrent attribute objects cannot interfere with one another, although a single attribute object is not safe against concurrent usage within a multi-threaded application; that is, problems can result from simultaneous *m_setlocale*( ) modifications of the same attribute object, concurrent with calling any multi-locale function dependent on that object. In this case, threads should use separate attribute objects to guard against outside interference.

Finally, an attribute object can be destroyed by calling *m_destroyattrobj*( ). This destroys the attribute object itself and any embedded objects, including the locale object.

3.4 Multi-locale Functions

This section provides an introduction to the multi-locale functions specified in the Distributed Internationalisation Services Specification. Rationales are included, where appropriate, describing reasons why interfaces emerged the way they have, alternatives that may have been considered, and any other background information of a relevant nature.

The interfaces are grouped into functional units, with each group being introduced by a table identifying:

- the global locale function or functions used as an archetype for the multi-locale function. This may be omitted if no global locale function defines equivalent capability.
- the name or names of the multi-locale functions
- an indication of how the global locale and multi-locale interfaces differ, in terms of differences in the multi-locale version.

New functions, for which there is no equivalent in the global locale model, are explained in more detail. Examples are provided as appropriate, mainly to illustrate possible uses of novel functions.

3.4.1 Locale Management Functions

These functions provide control over attribute objects, multi-byte state objects, and the orientation of I/O streams.

Global-locale	Multi-locale	Comments
setlocale( )	*m_createattrobj*( ) *m_destroyattrobj*( ) *m_setlocale*( )	(see above) (see above) **AttrObject** added
	m_creatembstate( ) *m_destroymbstate*( )	(see below) (see below)
	m_fattr( )	(see below)

Multi-byte State Objects

The *m_creatembstate*( ) function returns an object of type **mbstate_t**, placed in a codeset-dependent initial state, which is used to retain state information between successive calls of certain context-sensitive functions. For example:

```
AttrObject attrobj;
mbstate_t state;
char *str;
...
attrobj = m_createattrobj( );
str = m_setlocale(attrobj, LC_ALL, "");
state = m_creatembstate(attrobj)
...
while (p) {
    p = m_strtok (SomeString, "\t", state);
    SomeProcessing ( );
}
...
```

The encoding of *SomeString* may be single-byte, multi-byte or multi-byte with state encodings. In the latter case, it is necessary to retain shift state information between successive calls to the string tokenising function *m_strtok*( ). This is needed to allow other threads to use the function concurrently, or to allow multiple uses of *m_strtok*( ) to tokenise different strings.

Note that because an attribute object, and hence a locale object, can be associated with an **mbstate_t** object by means of the *m_creatembstate*( ) function, interfaces that accept an **mbstate_t** object as an argument do not usually have an explicit **AttrObject** argument as well.

The *m_destroymbstate*( ) function is used to destroy an **mbstate_t** object and takes no arguments other than the **mbstate_t** value returned by a previous successful call to *m_creatembstate*( ).

Stream Orientation

The **ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C** *fwide*( ) function can be used to determine the orientation of an I/O stream; that is, whether it is byte-oriented or wide-oriented. This function is defined as:

```
int fwide(FILE *stream,
          int mode);
```

If *mode* is greater than zero, the *fwide*( ) function first attempts to make *stream* wide-oriented. If *mode* is less than zero, the function first attempts to make the stream byte oriented.

Note that if the orientation of *stream* has already been determined, the *fwide*( ) function does not change it.

If *mode* is zero, the function does not change the orientation of *stream* and the return value from the call indicates its current orientation.

After a stream is associated with an external file, but before any operations have been performed on it, the stream is without orientation. Once a wide-character I/O function has been applied to a stream without orientation, the stream becomes wide-oriented. Similarly, once a byte I/O operation has been applied to a stream without orientation, the stream becomes byte-oriented. Only a call to the *freopen*( ) function or *fwide*( ) function can otherwise alter the orientation of a stream.

Wide-orientation does not automatically imply that code elements will be stored in the external file as wide-character codes. For example, an implementation may choose instead to employ a multi-byte encoding (for example, FSS-UTF[1]), as doing so would still allow the file to be read as a byte stream, with the correct line terminators and no embedded null bytes. When read as a wide-oriented stream, the implementation would then be responsible for converting between FSS-UTF and the locale process code, as determined by the locale object associated with the stream used to access the file. Conversely, a universal multiple-octet coded character set (UCS) implementation (ISO/IEC 10646 for example), might and probably would elect to store file codes as wide-character values directly.

The *m_fattr*( ) function is provided to associate an explicit **mbstate_t** object with an I/O stream, irrespective of its orientation. After such an association, the stream is capable of locale specific processing in a multi-locale environment.

The *m_fattr*( ) function is defined as:

```
int m_fattr (FILE *stream,
             const AttrObject attrobj);
```

A successful call to this function causes an **mbstate_t** object to be created as if *m_creatembstate(attrobj)* had been called. This object is then associated with the I/O stream identified by *stream*.

If the *m_fattr*( ) function is not called explicitly, a wide-oriented stream will have an implicit **mbstate_t** object associated with it. In this case, locale specific processing will be performed using the global locale.

A call to *m_fattr*( ) with *attrobj* set to (**AttrObject**)NULL can be used to query the **AttrObject** of the **mbstate_t** associated with the stream.

[1] File System Safe UCS Transformation Format, or UTF-8.

3.4.2 Locale Information Functions

These functions provide access to a locale object for querying the the settings of locale-specific information in the LC_TIME, LC_MONETARY, LC_NUMERIC and LC_MESSAGE categories.

Global-locale	Multi-locale	Comments
localeconv( )	*m_localeconv*( )	**AttrObject** added
nl_langinfo( )	*m_nl_langinfo*( )	(see below)
MB_CUR_MAX	*m_mb_cur_max*( )	(see below)
strerror( )	*m_strerror*( )	**AttrObject** added

The m_nl_langinfo( ) Function

In the global model, *nl_langinfo*( ) is defined to accept one argument, *nl_item*, and to return a pointer to a string containing the associated locale data. The area pointed to by this string is controlled by the *nl_langinfo*( ) function itself, should not be modified by the program and may be modified by subsequent calls to *nl_langinfo*( ). In addition, calls to *setlocale*( ) may also overwrite this area.

Such a definition is inherently unsafe in the world of object oriented, multi-layered or multi-threaded applications, and is another reason why the global locale model is unsuited to meet the needs of these applications. Hence, in the multi-locale model, the program itself is made responsible for storage management, as follows:

```
char *m_nl_langinfo(const AttrObject attrobj,
                    nl_item item,
                    char *buf,
                    size_t bufsize);
```

The *m_nl_langinfo*( ) function is defined to return locale-specific data, as identified by the value of *nl_item*, in the storage area pointed to by *buf*, of no more than *bufsize* bytes. The *m_nl_langinfo*( ) function returns a pointer to a null string if a call is unsuccessful.

Values returned in *buf* will be encoded in the codeset of the locale identified by *attrobj* as a byte stream. If a wide-character form of the data is required, the program itself is responsible for doing this by means of calls to the **ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C** function *mbsrtowcs*( ). Thus no wide-character form of the *nl_langinfo*( ) function is defined in the multi-locale API.

A new limit value is also introduced, MAX_INFO_MSG_LEN, which gives the maximum size of a string (in bytes) that can be returned by the *m_nl_langinfo*( ) function, including the terminating null byte. This value is added to **<limits.h>**.

The m_mb_cur_max( ) Function

The number of bytes that can be used to hold the characters of a locale is defined by the values of **<mb_cur_min>** and **<mb_cur_max>** in a Character Set Description File, which is input to the *localedef* utility when a locale is created. On XPG4 systems, **<mb_cur_min>** is always 1; **<mb_cur_max>** can be a value greater than or equal to 1 (which is the default).

The maximum number of bytes in a character in any supported locale is given by the value of the **<limits.h>** constant MB_LEN_MAX. In the global locale model, a program can determine the maximum number of bytes in any character of the current locale by means of the MB_CUR_MAX macro defined in **<stdlib.h>**, which is defined as an integer expression.

Clearly, this mechanism is not applicable to the multi-locale model, where the *mb_cur_max* value may vary between coexisting locales. Hence, a new function is defined that operates on a specific locale object, as follows:

```
int m_mb_cur_max(const AttrObject *attrobj);
```

which returns the maximum number of bytes in any character specified in the locale associated with *attrobj*.

Note that the issues of multi-byte codeset support and multi-locale support should not be confused. The requirement for multi-locale support applies even in territories where multi-byte support is not necessarily relevant. In Europe, for example, a system may be delivered with a set of locales for the European languages (using ISO 8859 encodings), none of which support multi-byte operation, but which an application may still require to manage concurrently. Indeed, neither XPG4 nor the **Distributed Internationalisation Services** snapshot do or can mandate support for multi-byte codesets, which is a business decision on the part of individual vendors.

3.4.3 Classification Functions

These functions provide for the classification of code elements.

Global-locale	Multi-locale	Comments
wctype( ) *iswctype*( )	*m_wctype*( ) *m_iswctype*( )	**AttrObject** added **AttrObject** added
	m_isctype( ) *m_strscanfor*( ) *m_wcsscanfor*( )	(see below) (see below) (see below)

The *is**( ) and *isw**( ) functions of the global locale model are not reproduced in the multi-locale model. These interfaces increase the size of the API without providing any extra capability. Also their capability is easily reproduced using the *m_iswctype*( ) function. For example:

```
iswalnum (wc);
```

is equivalent to:

```
m_iswctype(attrobj, wc, m_wctype (attrobj, "alnum"));
```

when the current locale is set to the locale defined by *attrobj*.

The m_isctype( ) Function

This function is similar to the *m_iswctype*( ) function, except that it provides for the classification of code elements in character strings, rather than wide-characters. Thus:

```
int m_isctype(const AttrObject attrobj,
              char *s,
              wctype_t desc);
```

The *m_isctype*( ) function determines whether the first coded character in the string pointed to by *s* has the character class of *desc*. For example:

```
...
wctype_t desc = m_wctype(attrobj, "space");
mbstate_t ps;
char *s;
...
ps = m_creatembstate(attrobj);
while(*s && m_isctype(attrobj, s, desc))
    s += mbrlen(s, strlen (s), &ps);
...
```

This code will step along the characters in the string pointed to by *s*, until a non-space character is found or a null character is detected. Note that the *m_mblen*( ) function allows state information to be retained between successive calls by means of the **mbstate_t** object *ps*.

The only thing that could interfere with the integrity of this code is if the locale associated with *attrobj* were changed or modified between the *m_wctype*( ) and *m_isctype*( ) calls. This danger could be obviated by changing the *m_isctype*( ) call to:

```
m_isctype(attrobj, s, m_wctype (attrobj, "space"));
```

But this might prove costly in the context of the above code sequence.

The m_*scanfor( ) Functions

The *m_*scanfor*( ) functions provide another dimension to character classification not present in the standard API. These functions allow a coded character string or a wide-character string to be scanned for the presence or non-presence of a particular character type, as follows:

```
size_t m_strscanfor(const AttrObject attrobj,
                    const char *s,
                    size_t num_chars,
                    size_t position,
                    ScanDirection direction,
                    ScanCondition condition,
                    Boolean inverse);

size_t m_wcsscanfor(const AttrObject attrobj,
                    const wchar_t *ws,
                    size_t num_chars,
                    size_t position,
                    ScanDirection direction,
                    ScanCondition condition,
                    Boolean inverse);
```

where the **Boolean** type is defined as:

```
typedef enum {False, True} Boolean;
```

The *m_strscanfor*( ) (*m_wcsscanfor*( )) function scans the coded character (wide-character) string pointed to by *s* (*ws*), starting at the offset indicated by *position*, for the first coded character (wide-character) code that matches or does not match the classification criteria specified by *condition*. If *inverse* is set to False, the function searches for the first coded character (wide-character) that matches the search criteria. Otherwise, if *inverse* is True, it searches for the first coded character (wide-character) that does not match the search criteria.

The *direction* argument determines the direction in which scanning takes place, and is an integer type defined as:

```
typedef enum {Forw, Back} ScanDirection;
```

If *direction* is Forw, the function scans from *position* to the end of the coded character (wide-character) string. If *direction* is Back, the functions scans from *position* to the beginning of the coded character (wide-character) string.

The *condition* is defined as a bitwise OR of one or more classification criterion specified in **<mlocale.h>**. Using the *m_wcsscanfor*( ) interface, scanning for a non-space wide-character, similar to the example given in the previous section, could be rewritten as:

```
...
wchar_t *ws;
size_t offset;
...
    offset = m_wcsscanfor (attrobj, ws, wcslen (ws),
                 0, Forw, WhiteSpace, True);
...
```

There is a *condition* criterion associated with each of the standard character classes, plus some more esoteric criterion for detecting such things as line breaks, word, sentence and paragraph boundaries, and so on. Consult the Distributed Internationalisation Services Specification for more details.

3.4.4 Transliteration Functions

These functions provide conversion operations similar to, but more comprehensive than, the *to**( ) and *tow**( ) conversion functions defined in the standard API.

Global-locale	Multi-locale	Comments
	m_wctrans( )	(see below)
	m_tombstrans( )	(see below)
	m_towcstrans( )	(see below)

These functions work together in a manner similar to the *m_wctype*( ) function and the *m_is[w]ctype*( ) functions described in the previous section, where the *m_wctrans*( ) function is defined as:

```
wctrans_t m_wctrans(const AttrObject attrobj,
                    const char *property);
```

The *property* is character string identifying the required transliteration. Only *upper* and *lower* are defined in all locales; additional transliteration names may by defined in a locale definition file for category LC_CTYPE.

```
int m_tombstrans (const AttrObject attrobj,
                  wctrans_t desc,
               char **inbuf,
               size_t *inbufleft,
               char **outbuf,
               size_t *outbufleft);

int m_towcstrans(const AttrObject attrobj,
                 wctrans_t desc,
                 wchar_t **inbuf,
                 size_t *inbufleft,
                 wchar_t **outbuf,
                 size_t *outbufleft);
```

These functions transform the characters in the multi-byte (wide-character) string *inbuf*, as directed by the setting of *desc*, and write the results to the multi-byte (wide-character) string *outbuf*.

On input, *inbufleft* specifies the number of code elements to be transformed, or -1, which indicates that the input is delimited by a null code element. On return, the value pointed to by *inbufleft* will contain the number of code elements still to be processed.

On input, *outbufleft* specifies the size of the output buffer in terms of bytes (wide-character codes). On return, the value pointed to by *outbufleft* will contain the actual number of bytes (wide-character codes) placed in the output buffer. If this value is zero on input, no transformation takes place and the size of the *outbufleft* needed to transform the contents of *inbuf* is returned.

For example:

```
wchar_t      inbuf [BUFSIZ];
wchar_t outbuf [BUFSIZ];
wctrans_t desc;
size_t ibl, obl;
...
desc = m_wctrans (attrobj, "lower");

while (fgetws (inbuf, BUFSIZ, Istream)) {
       ibl = -1, obl = BUFSIZ;
       m_towcstrans (attrobj, desc, &inbuf, &ibl,
              &outbuf, &obl);
       fputws (outbuf, Ostream);
}
...
```

This will transform the contents of the file referenced by *Istream* to lower case and write the output to the file referenced by *Ostream*. All case conversion will be done according to the LC_CTYPE category of the locale associated with *attrobj*.

3.4.5 String Searching Functions

These functions provide facilities for searching code element strings for specific code elements or substrings.

Global-locale	**Multi-locale**	**Comments**
strpbrk( )	*m_strpbrk*( )	**AttrObject** added
strspn( )	*m_strspn*( )	**AttrObject** added
strcspn( )	*m_strcspn*( )	**AttrObject** added
strstr( )	*m_strstr*( )	**AttrObject** added
wcspbrk( )	*m_wcspbrk*( )	**AttrObject** added
wcsspn( )	*m_wcsspn*( )	**AttrObject** added
wcscspn( )	*m_wcscspn*( )	**Attrobject** added
wcswcs( )	*m_wcswcs*( )	**AttrObject** added

These functions closely match the standard functions defined in XPG4, except that searching is done according to the locale object associated with *attrobj*, rather than the global locale. Also comparisons performed by the *m_str**( ) functions are only done on complete coded characters and at coded character boundaries. Thus, in locales that support multi-byte encodings, comparisons will never be done on incomplete character codes.

3.4.6 String Comparison Functions

These functions provide for the comparison of code elements within code element strings.

Global-locale	**Multi-locale**	**Comments**
strcoll( ) *strxfrm*( )	*m_strcoll*( ) *m_strxfrm*( )	**AttrObject** added **AttrObject** added
wcscoll( ) *wcsxfrm*( )	*m_wcscoll*( ) *m_wcsxfrm*( )	**AttrObject** added **AttrObject** added

The multi-locale string compare functions behave the same as the global locale functions when called with a locale object equal to the current global locale.

3.4.7 Date, Monetary and Time Formatting Functions

These functions provide for conversion to or from locale-specific monetary, date and time code element strings.

Global-locale	Multi-locale	Comments
strftime( )	*m_strftime*( )	**AttrObject** added
strptime( )	*m_strptime*( )	**AttrObject** added
strfmon( )	*m_strfmon*( )	**AttrObject** added
wcsftime( )	*m_wcsftime*( )	**AttrObject** added
	m_wcsptime( )	(see below)
	m_wcsfmon( )	(see below)

The wcsptime( ) Function

The *m_wcsptime*( ) function is added to the API to provide a wide-character equivalent of the *m_strptime*( ) function.

```
wchar_t *m_wcsptime(const AttrObject attrobj,
                    const wchar_t *ws,
                    const char *format,
                    struct tm *timptr);
```

This function converts the wide-character string pointed to by *ws* to a **tm** structure pointed to by *timptr*, using the format specified by *format*.

The format string is encoded in the file code of the locale associated with *attrobj*.

The wcsfmon( ) Function

The *wcsfmon*( ) function provides a wide-character equivalent to the *m_strfmon*( ) function:

```
size_t m_wcsfmon(const AttrObject attrobj,
                 wchar_t *ws,
                 size_t maxsize,
                 const char *format,
                ...);
```

The *m_wcsfmon*( ) function behaves the same as the *strfmon*( ) function when called with the current locale set to the locale of *attrobj*, except that a wide-character string is produced instead of a multi-byte string.

3.4.8 Number Conversion Functions

These functions provide conversion between code element strings and internal number representations.

Global-locale	Multi-locale	Comments
strtod( ) *strtol*( ) *strtoul*( )	*m_strtod*( ) *m_strtol*( ) *m_strtoul*( )	**AttrObject** added **AttrObject** added **AttrObject** added
wcstod( ) *wcstol*( ) *wcstoul*( )	*m_wcstod*( ) *m_wcstol*( ) *m_wcstoul*( )	**AttrObject** added **AttrObject** added **AttrObject** added

As can be seen, the only difference between these functions and their global locale counterparts is the addition of an attribute object.

3.4.9 Text Scanning and Parsing Functions

These functions allow applications to scan and parse localised code element strings into tokens.

Global-locale	Multi-locale	Comments
strtok( )	*m_strtok*( )	**mbstate_t** added
wcstok( )	*m_wcstok*( )	**mbstate_t** added

In all cases an **mbstate_t** object is added to the argument list. This allows state information to be retained between calls to the function, and also associates an attribute object with the call. In all other respects, these functions are equivalent to their global locale counterparts.

3.4.10 Formatted I/O Functions

The **Distributed Internationalisation Services** snapshot provides the *printf*( ) and *scanf*( ) family of functions with the capability of supporting various classes of encodings, including single-byte, multi-byte, stateful and contextual codesets.

Any locale-specific processing done within a stream depends on the stream having an **mbstate_t** object associated with it, which stores the current state of the stream.

Although both text and binary wide-oriented streams are conceptually sequences of code elements, the external file associated with a wide-oriented stream is a sequence of multi-byte characters, generalised as follows:

- encodings within files may contain embedded null bytes (unlike internal multi-byte encodings)
- a file need not begin or end in the initial state. Moreover, the encodings used for encoding text in internal files may differ between files. An **mbstate_t** object allows different streams to be associated with different locale objects having different encodings. Both the nature and the choice of such encodings are implementation defined.

The following functions may be used in a multi-locale environment when the streams **mbstate_t** object is associated with a specific locale object:

fwprintf( ) *swprintf*( ) *fwscanf*( ) *swscanf*( )
fgetpos( ) *fsetpos*( )
fgetwc( ) *fgetws*( ) *fputwc*( ) *fputws*( ) *ungetwc*( )

All locale specific processing (conversion, parsing, numeric formatting) is associated with the locale object contained in the stream's associated **mbstate_t** object.

A successful call to the *fgetpos*( ) function stores a representation of the value of the stream's **mbstate_t** object as part of the value of the **fpos_t** object. A later call to *fsetpos*( ) using the same value of **fpos_t** will restore the associated **mbstate_t** object as well as the position within the stream.

A number of global locale functions may also be used to access I/O streams in a multi-locale environment, but when and only when the current locale matches the locale associated with a stream's **mbstate_t** object (see the **Distributed Internationalisation Services** snapshot for details).

3.4.11 Extended Wide-character Conversion Functions

These functions allow code element strings to be converted from one type of encoding to another; for example, from multi-byte character strings to wide-character strings, and vice versa.

Global-locale	Multi-locale	Comments
mblen( )	*mbrlen*( )	**mbstate_t** added
mbtowc( )	*mbrtowc*( )	**mbstate_t** added
mbstowc( )	*mbsrtowcs*( )	**mbstate_t** added
wctomb( )	*wcrtomb*( )	**mbstate_t** added
wcstombs( )	*wcsrtombs*( )	**mbstate_t** added

Compared to their global locale counterparts, each of these functions has an additional argument of type **mbstate_t**, used to hold state and locale information between successive calls. Their are also a number of other differences that should be noted:

- For *mbsrtowcs*( ), the input string argument *src* is defined as a **char**** rather than a *char**. On return, and assuming the output wide-character string argument *dst* is not null, the object pointed to by *src* will contain either a null pointer (processing stopped due to reaching a terminating null character), or the address just past the last coded character converted (if any).
- For *mbsrtowcs*( ), if the output wide-character string pointer *dst* contains the null wide-character pointer, the function returns the number of elements required to hold the converted code element string.
- For *wcsrtombs*( ), the input wide-character string *src* is defined as a **wchar_t**** rather than a *wchar_t**. On return, and assuming the output string argument *dst* is not null, the object pointed to by *src* will contain either a null wide-character pointer (processing stopped due to reaching a terminating null wide-character), or the address just past the last wide-character code converted (if any).
- For *wcsrtombs*( ), if the output character string pointer *dst* contains the null pointer, the function returns the number of elements required to hold the converted code element string.

For all the above functions, if the **mbstate_t** object refers to the global locale, the function behaves as defined in the **ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C** (see Appendix A). If the **mbstate_t** object refers to a valid locale object, conversions are performed as defined by that locale object rather than the global locale. In this way, these functions can operate either in a global locale or a multi-locale environment.

Chapter 4

Advanced Text Handling

This chapter considers the requirements of advanced text handling in detail, including state handling requirements in a multi-locale environment, context sensitive and bidirectional text handling.

4.1 Background

The XPG4 specification perpetuates the ISO and ANSI paradigm of presenting basic text entities that are closely associated with a glyph or display cell; that is, one coded character in the underlying codeset is assumed to equal one display character. There is no provision for dealing with, or even recognising, the existence of composite sequences, where multiple text elements are combined to form a single character for the purposes of collation, character classification, passing through the I/O system, and so on.

Both ISO 6937 and ISO/IEC 10646 define floating diacritics, which are context sensitive characters that may or may not be combined with a preceding base character to form a composite sequence. To satisfy this requirement necessitates functions that can handle characters represented by composite sequences, just as the XPG4 specification was enhanced to handle multi-byte characters.

Text directionality presents further problems in this area. Certain languages require that text objects can be processed either left-to-right or right-to-left, depending on the character class of the text object (letters as opposed to numbers). Directionality can change in mid-string and should be correct in both the processing case and the presentation case.

Another issue arises with the text shaping aspects in scripts of languages such as Arabic. When presented, each character may assume one of up to four different shapes, depending on the context; that is, the connectivity characteristics of the character itself and of its neighbour characters. In most of the cases, mainly because of keyboard limitations when entering data and also because of lack of enough code points in some encoding schemes, such texts are processed with all characters in a *basic* shape, which has to be *shaped* by a shaping transformation before the text is presented.

These issues need to be addressed when considering new APIs associated with program internationalisation. It is too late to retrofit this capability to the XPG4 API, but it would be remiss to propose a multi-locale API that still made no provisions for handling composite sequences and text directionality.

4.2 Complex Text

First, some terms commonly used when referring to this subject:

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.

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.

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.

complex-text languages
A collective name used to designate those languages that have different layouts for processing the text and for presenting it. The complex-text languages include the bidirectional languages (such as Arabic, Farsi, Urdu, Hebrew, Yiddish), and Asian languages such as Thai, Lao, Korean and the Indian ones. Because they are dealt with separately, the languages that use mainly an ideographic script, such as Chinese and Japanese, are excluded from this definition.

4.2.1 Codesets

Commonly used codesets and encoding methods, such as ASCII, ISO 8859-1, and Japanese EUC, include characters for a single language (writing system) 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. The goal in creating ISO 10646 was to include all characters from all significant languages; that is, to be what the standard calls a ‘‘Universal Multi-Octet Coded Character Set’’ (UCS). The initial version of 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 10646 specifies the same codeset as Unicode (1.1)[2]. Unicode was developed, and is maintained by, the Unicode Consortium, which has a wide membership base of information technology companies. People often use ‘‘10646’’ and ‘‘Unicode’’ interchangeably, although

[2] Technically speaking, Unicode (1.1) in BMP is a usage profile of ISO 10646.

there are differences between the two sets.

ISO 10646 differs in some ways from codesets currently used on XPG-compliant systems. Many currently supported codesets include portable characters as single-octet entities and with code values matching either ISO 646 IRV:1991 or a form of EBCDIC. The ISO 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 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 10646, in contrast, are encoded in multiple octets. Code space is organised into 128 groups of 256 planes each.

10646 allows two basic forms for characters:

- 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 10646.
- Universal Coded Character Set-4 (UCS-4). Characters are encoded in the full four octets.

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

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

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

4.2.2 Composite Sequences

In addition to the UCS-2 and UCS-4 forms, ISO 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 and Arabic National Standards.

For example, suppose you want to encode the letter <a-acute> (lowercase a with acute accent). This letter-with-diacritic exists in ISO 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 10646, certain characters are defined as *combining marks*; 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 10646:

```
+--------+--------+--------+--------+-------+
|   p    |   ´    |   ˜    |   ˆ    |   `   |
+--------+--------+--------+--------+-------+
```

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

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

Although combining characters give ISO 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 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 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 code elements are two octets each. Unicode has no concept of levels and is equivalent to UCS-2, Level 3.

ISO 10646 is only used here as an example of a codeset that supports combining sequences. A multi-locale compliant system that required to support (say) UCS-2, Level 3 or Unicode R1.1 could do so by defining these codesets as the process code in appropriate locales; that is, where the size of **wchar_t** would be at least two bytes and *m_mb_cur_max*( ) would return a value of 2. Representation on file storage could be either directly as wide characters, or encoded in one of the Universal Transformation Formats (for example, UTF-8). Either way, to fully support these codesets, the implementation would also require to support composite sequences.

Certain multi-locale functions are therefore defined to work correctly in locales that support composite sequences, including:

- the *m_wcsscanfor*( ) function, for classification
- the *m_towcstrans*( ) function, for transliteration.

In addition, a completely new set of functions are included in the API to operate on code element strings that may contain composite sequences. These functions are described in detail later in this chapter.

4.3 Directional and Context-sensitive Text

This refers to languages with a script that is directional or which may have different shaping forms for its characters. From a directional point of view, the text is said to be in a physical order when presented on a presentation device such as a printer or a display. In physical order, some parts of the text (called segments) may appear to be written in a direction opposed to the general direction of the text. For some languages, the shapes of the characters, when presented on a presentation device, may differ according to their ability to connect with neighbouring letters. As opposed to the physical order, text strings of such scripts have a logical order that is the order in which the coded elements are pronounced when read or the order in which the text is usually entered. In those languages whose script has shaping forms, the logical stream may contain coded elements that have an encoding with a shape different to the shape rendered on a presentation device. Sometimes the logical character streams may contain directional control coded elements and shaping control coded elements that may be used in order to transform between the logical stream and the physical stream. In many cases the transformation between the logical stream and the physical stream is done based not on embedded controls but on the context of the coded elements. Hence the term ‘‘context-sensitive text’’.

Once these external encodings are mapped to a code element string, directional introducers are viewed as any other code elements that are primarily intended for use by presentation services. Hence, functions based on a locale object treat these encodings just like any other code element within a text string.

The important point being made here is that the multi-locale functions will only process text strings correctly if they are presented in logical order. For example, any text that is in *visual* order (the order presented externally) needs to be transformed into logical order before any collation function can operate on the text.

Functions for transformation between the logical and visual ordering of text are not currently presented in the multi-locale API. They are covered in the **Layout Services** snapshot. Within the current definition, such introducers are not excluded but they are not catered for implicitly by the multi-locale functions. Work is continuing in this and related areas.

4.4 Composite Sequence Functions

These functions are defined to work on strings that may consist of either a composite sequence or a single code element.

Global Locale	Multi-locale
	m_wcscnt( )
	m_wcsnext( )
	m_wcsquery( )
wcwidth( )	
wcswidth( )	*m_wcswidth*( )

The m_wcscnt( ) Function

This function counts the number of wide-character codes in a composite sequence.

```
size_t m_wcscnt(const AttrObject attrobj,
                const wchar_t *ptr);
```

If *ptr* points to a non-combining wide-character, this function returns the number of wide-character codes making up the addressed code element, including the non-combining character and all following combining characters (if any). If *ptr* points to a null wide-character or a combining character, *m_wcscnt*( ) returns zero.

The m_wcsnext( ) Function

This function locates the next non-combining wide-character in a wide-character string.

```
size_t m_wcsnext(const AttrObject attrobj,
                 const wchar_t *ptr);
```

If *ptr* points to a null wide-character or a combining character, this function returns zero. Otherwise, if *ptr* points to a non-combining wide-character, it returns the offset to the start of the next non-combining wide-character in the string.

The m_wcsquery( ) Function

This function determines the number of composite sequences in a wide-character string.

```
size_t m_wcsquery(const AttrObject attrobj,
                  const wchar_t *ptr);
```

The *m_wcsquery*( ) function will return zero if either *ptr* points to a null wide-character, or the wide-character string referenced by *ptr* does not contain any composite sequences (that is, a non-combining character followed by one or more combining characters). Otherwise, *m_wcsquery*( ) returns the number of composite sequences present in the string.

The m_wcswidth( ) Function

This function determines the width of a composite sequence in terms of column positions.

```
size_t m_wcswidth(const AttrObject attrobj,
                  const wchar_t *ptr,
                  size_t n);
```

It is assumed that graphic symbols are a fixed presentation column width, where the column width of any graphic symbol is an integral multiple of a unit column width graphic symbol.

The *m_wcswidth*( ) function returns the number of column positions required by *n* composite sequences in the wide-character string pointed to by *ptr*. If *ptr* points to a null wide-character or a combining character, the function returns zero.

Chapter 5

Distributed Internationalisation

This chapter deals with the requirements and proposed solutions for internationalisation within distributed applications. This applies both to local distribution within the same system and distribution in a hetrogeneous network.

As an example, for a client-server application to provide consistent behaviour in both its client-part and its server-part, it must be able to replicate locale-specific behaviour in both parts. This requires that:

- it must have some means of identifying the required locale
- it must have a mechanism for re-creating the locale
- there must be a common definition of locales that all systems in the network understand.

This misleadingly simple statement of requirements gives rise to a whole set of problems and issues not addressed by existing standards. These and solutions proposed by a combination of the **Distributed Internationalisation Services** snapshot and procedures for locale registration are discussed in the following sections.

5.1 The Problem

Consider the the requirements of a remote procedure call (RPC) mechanism that must export its calling locale to the server environment. It would probably want to do something like the following:

1. announce to the server the locale of its calling environment
2. in the server, either accept the clients locale, if it can be re-created locally, or negotiate for some other common locale, or fail if no common working method can be established
3. negotiate a common encoding and data type.

The first problem to surface is one of naming. Neither XPG4, POSIX nor ISO C define standard locale names, other than for the default C and POSIX locales. XPG4 does include a naming syntax as follows:

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

However, it does not define semantics of the element settings nor any standard values. No specific codeset support is mandated either. Collation and character classification rules are defined in the C or POSIX locale (ASCII), and the minimum character set is also specified (the Portable Character Set).

Secondly, not only are there no standard names for locales. there are no standard locale definitions either. By long tradition and common practise, most implementations provide a US ASCII locale that supports US English, US customs and either ASCII or some 8-bit codeset that incorporates the ASCII codeset within it (for example, ISO 8859-1). This is a long way from standard locale definitions of a type required by client-server applications.

Next come problems concerned with encodings and data types. There are no standard encodings mandated for XPG, POSIX or ISO C conformance and the size and orientation of data types is implementation-specific. For example, XPG4 defines that **wchar_t** can be an integral value of any size, including a single-byte. This may sound incongruous, but the supply of machines in Europe (for example) does not obviously require multi-byte support, so one or more 8-bit codesets is probably adequate.

At an even more fundamental level, the size of a *byte* and the *char* type are not fixed either. 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. It further defines that a byte is composed of a contiguous sequence of bits, the length of which is implementation-defined. An object of type **char** is defined to be large enough to store any member of the basic execution set (minimum 8-bits).

Being optimistic and assuming the above problems could be overcome somehow, there are higher-level issues that also need to be considered. Imagine, for example, a multi-threaded server trying to cope with multifarious RPC requests from a host of clients, all having different localisation requirements. The poor old global locale would never cope, although of course we already have an answer to this problem in the form of the multi-locale model. Indeed, the multi-locale model is fundamental to distributed internationalisation, which is why the previous chapters of this document have been almost exclusively devoted to describing it, and why distributed internationalisation (the subject of this document) has not been mentioned until now.

The multi-locale functions enable a server of the type described above to manage locale and possibly state information on a per client basis. This also works if the server is multi-layered or object oriented.

However, that does still leave one or two other not insignificant problems to be solved, namely:

- locale naming conventions
- common locale definitions
- some way of exporting locale specifications in a network-independent manner.

5.2 Locale Naming

The **Distributed Internationalisation Services** snapshot introduces the concept of a network locale specification, which is an abstraction used for representing the name of a particular locale as a network object. On a host system, a network locale specification is encapsulated in a program object of type **LocaleSpec**. On a network, a network locale specification may be represented as a network locale specification token of type **LocaleNetToken**, or as a network locale specification string of type **LocaleNetString**.

Functions are provided for creating and destroying **LocaleSpec**s, for converting between host locale specifications and network locale specifications, and for mapping between host **LocaleSpec**s and network independent **LocaleNetToken**s and **LocaleNetString**s (see later in this chapter for further details).

5.2.1 String Network Locale Specifications

A network locale specification string provides a name for each category that exists within a given locale, although not all categories from the standard list must be present. The network locale specification string does not specify LC_ALL, which may refer to locales containing optional categories and is therefore ambiguous, but calls out each specific locale category instead. For example:

```
CTYPE=ANSI;en_US;01_00;XFN-001001;/
COLLATE=ANSI;en_US;01_00;XFN-001001;/
MESSAGES=ANSI;en_US;01_00;XFN-001001;/
MONETARY=ANSI;en_US;01_00;XFN-001001;/
NUMERIC=ANSI;en_US;01_00;XFN-001001;/
TIME=ANSI;en_US;01_00;XFN-001001;/
```

where:

CTYPE
COLLATE
MESSAGES
MONETARY
NUMERIC
TIME
Indicates the locale category to which the line relates.

ANSI
Indicates the registration authority for the locale specification.

en_US
Indicates the locale name.

01_00
Indicates the version and revision number of the locale specification.

XFN-001001
Indicates the registered encoding defined in the **Federated Naming** specification.

The full syntax of string network locale specifications is given in the **Distributed Internationalisation Services** snapshot. For the purposes of this document, it is sufficient to know that such a string can uniquely identify a locale and its contents.

5.2.2 Token Network Locale Specifications

The size and complexity of a network locale specification string may be prohibitive in terms of size or performance. Each locale specification, therefore, may also have associated with it a four octet, unsigned integer value known as a Network Locale Specification Token. A token for a given locale implies that all standard categories have the same locale name.

The four octets of a network locale specification token can broken up into two parts:

- the first two octets denote the registration authority for the token
- the last two octets give the individual token number.

This scheme permits 65,535 registration authorities, each with a registration space of 65,535 individual tokens.

5.2.3 LocaleSpec Functions

These functions provide for the generation and manipulation of host-dependent network locale specifications (**LocaleSpec**s).

LocaleSpec Functions
m_createlocspec( )
m_locspec_to_host( )
m_locspec_from_host( )
m_destroylocspec( )

The *m_createlocspec*( ) function, defined as:

```
LocaleSpec *m_createlocspec(void);
```

returns an object of type **LocaleSpec** that contains an empty network locale specification. A **LocaleSpec** is an object type other than an array type that can hold values representing a locale specification. Although the content of a *LocaleSpec* is opaque to an application, it can be thought of as consisting of a **LocaleNetToken** or **LocaleNetString**, and is the principle object used by client-server programs for announcing locales in a distributed environment.

A **LocaleSpec** object is released by calling the *m_destroylocspec*( ) function:

```
int m_destroylocspec(LocaleSpec *locspec);
```

The *m_locspec_to_host*( ) and *m_locspec_form_host*( ) functions are used to convert between a network locale specification and a host locale specification of the type used for calls to *setlocale*( ) and *m_setlocale*( ).

```
char *m_locspec_to_host(const LocaleSpec locspec);

int m_locspec_from_host(LocaleSpec *locspec,
                        const char *s);
```

Thus a client application might use these functions to generate locale specifications as follows:

```
void foo_operation(AttrObject client_locale,
                   CLIENT     *client_handle)
{
    LocaleSpec      LocSpec;

    LocSpec = m_createlocspec( );
    if (m_locspec_from_host(LocSpec,
           m_setlocale(client_locale, LC_ALL, NULL))
       == -1) {

       /*
        *  Client’s locale is not representable as
        *  a locale specification.  Return an error
        */

       return;
    }

    rpc_foo_operation(LocSpec, client_handle);
    m_destroylocspec(LocSpec);
}
```

The *client_handle* is assumed to be some RPC-specific object that uniquely identifies each client with current access to the RPC server.

The call to *m_setlocale*( ) with the *locale* argument set to NULL, and *category* set to LC_ALL, causes the current locale settings of all locale categories associated with the locale identified by *client_locale* to be returned. This is then converted to a network locale specification (*LocaleSpec*) by the call to *m_locspec_from_host*( ), and passed across the network to a matching remote server function.

On the server side, the *rpc_foo_operation*( ) function can re-establish the client’s locale as follows:

```
rpc_foo_operation(localeSpec LocSpec,
                  struct svc_reqd *req)
{
    char     *host_spec;
    AttrObject      host_locale;

    if ((host_spec = m_locspec_to_host (LocSpec))
        == NULL) {

       /*
        *  Locale is NOT known to the server.
         *  return an error.
        */
    }

    host_locale = m_createattrobj( );

    if (m_setlocale(host_locale, LC_ALL, host_spec)
        == NULL) {

       /*
        *  Locale is NOT supported by the server.
        *  Return an error.
        */
    }
    free(host_spec);
    m_foo(host_locale);    /* do real work */
    m_destroyattrobj(host_locale);
}
```

5.2.4 NetSpec Functions

These functions provide for conversion between host-dependent network locale specifications of type **LocaleSpec** and network-independent network locale specification strings of type **LocaleNetString** and network locale specification tokens of type **LocaleNetToken**.

NetSpec Functions
m_locspec_to_nettoken( )
m_locspec_to_netstring( )
m_locspec_from_nettoken( )
m_locspec_from_netstring( )

As already stated, a **LocaleSpec** is a host-dependent, opaque data type that cannot safely be communicated over a network. The *m_locspec_to_nettoken*( ) and *m_locspec_to_netstring*( ) functions are provided to convert a locale specification to a form that may be communicated over a network. Conversely, the *m_locspec_from_nettoken*( ) and *m_locspec_from_netstring*( ) functions are provided to convert from a network format (token or string) to a locale specification.

These functions are intended for use by communication software needing to manage network independent data mappings. The *m_locspec_to_**( ) functions are expected to be used by client software (for example) when transmitting locale requirements, and the *m_locspec_from_**( ) functions might be used by server software to re-establish a locale in the server environment.

5.3 Locale Registration

Currently, there is no way to specify portably which locale a user or application requires to use. There are no standards for naming locales, other than the simple C and POSIX locales, and there is no common agreement about which locales refer to what language, territory and codeset combinations. Without such agreement, the distributed locale model described in the previous section is clearly untenable.

The main aim of the X/Open Locale Registry is to obviate this shortcoming by providing a repository where locales can be registered by appropriate bodies and implemented by suppliers for distribution with their products. A locale registry, which both defines unique names and the contents of locales, is vital in a distributed environment so that locale-sensitive operations on different systems can guarantee to produce the same results.

The stated goals of the X/Open Locale Registry are therefore:

- to guarantee the same behaviour (of APIs and utilities) across different systems, with respect to locale-sensitive operations
- to resolve the namespace collisions that can currently occur with respect to the naming of locales by different suppliers
- to support locale-sensitive operations in a hetrogeneous network of computers
- to provide an initial set of locale definitions that support the above goals
- to promote the migration of this registry into the appropriate JTC1 authority
- to encourage the careful growth of the registry through consensus.

Within this broad set of guidelines, it is not mandated that X/Open members must supply any published locale specifications, nor is it an aim to register every conceivable locale from every conceivable source. Which locales are made available on any particular system is a business decision for the supplier, although locales with registered names should produce predictable results on any system.

The means by which the Locale Registry was established, the acceptance criteria for locale submissions, and the syntax for locale specifications are documented in the **Locale Registry Procedures**. Readers are referred to this document for further information.

Appendix A

ISO MSE Changes

ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C, more familiarly known as the ISO MSE, proposes a set of library extensions that provide a complete and consistent set of utilities for application programming using multi-byte and wide characters. Much of what is proposed in the **ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C** is similar to extensions proposed in the **Distributed Internationalisation Services** snapshot, with the one important difference that the **ISO/IEC 9899:1990 Amendment 1:1994, Multibyte Support Extensions for ISO C** only addresses extensions to the global locale model.

The ISO MSE working group and the X/Open-Uniforum Joint Internationalisation Working Group have been cognizant of one another's activities, and there have been joint meetings to attempt to address differences between the two proposed sets of interfaces. Thus many of the principles are common, and there is even recognizable similarities in terms of function and type names, interface layout, and so on.

This appendix provides a brief comparison of the two sets of interfaces, highlighting both similarities and differences. It should also be assumed that a future issue of the Portability Guide will be aligned with the ISO MSE for global locale working.

A.1 Data Types

The ISO MSE and the **Distributed Internationalisation Services** snapshot define that the following data types are defined in the header **<wctype.h>**,

wint_t
wctrans_t
wctype_t

and that the object type **mbstate_t** is added to **<wchar.h>**. All other **Distributed Internationalisation Services** snapshot types, constants, macros and function prototypes are included in the header **<mlocale.h>**. None of these are present in the MSE. In particular, the MSE has no equivalent to the object types **AttrObject**, **LocaleSpec**, **LocaleNetToken** or **LocaleNetString**.

A.2 Functions

In many cases, a multi-locale equivalent to an MSE function is obtained simply by adding *m_* to the MSE name. For example, *wctrans*( ) in the MSE is equivalent to *m_wctrans*( ) in the **Distributed Internationalisation Services** snapshot, although the argument lists of an *m_* function reflect multi-locale operational requirements.

The following table lists the MSE functions, followed by an indication of equivalent functions in XPG4 and the **Distributed Internationalisation Services** snapshot (DISS). Again it should be remembered that while DISS functions may appear similar to MSE functions, they are defined to work in a multi-locale rather than a global locale environment. Hence they are normally defined to accept a locale object argument of some kind (either associated with an **AttrObject** or an **mbstate_t** object).

MSE	XPG4	DISS
iswalnum( )	*iswalnum*( )	
iswalpha( )	*iswalpha*( )	
iswcntrl( )	*iswcntrl*( )	
iswdigit( )	*iswdigit*( )	
iswgraph( )	*iswgraph*( )	
iswlower( )	*iswlower*( )	
iswprint( )	*iswprint*( )	
iswpunct( )	*iswpunct*( )	
iswspace( )	*iswspace*( )	
iswupper( )	*iswupper*( )	
iswxdigit( )	*iswxdigit*( )	
towlower( )	*towlower*( )	
towupper( )	*towupper*( )	
wctrans( )		*m_wctrans*( )
towctrans( )		*m_towctrans*( )
fwprintf( )†		
fwscanf( )†		
wprintf( )		
wscanf( )		
swprintf( )†		
swscanf( )†		
vfwprintf( )		
vwprintf( )		
vswprintf( )		
fgetwc( )	*fgetwc*( )	
fgetws( )	*fgetws*( )	
fputwc( )	*fputwc*( )	
fputws( )	*fputws*( )	
getwc( )	*getwc*( )	
getwchar( )	*getwchar*( )	
putwc( )	*putwc*( )	
putwchar( )	*putwchar*( )	
ungetwc( )	*ungetwc*( )	
fwide( )		

MSE	XPG4	DISS
wcstod( )	*wcstod*( )	*m_wcstod*( )
wcstol( )	*wcstol*( )	*m_wcstol*( )
wcstoul( )	*wcstoul*( )	*m_wcstoul*( )
wcscpy( )	*wcscpy*( )	
wcsncpy( )	*wcsncpy*( )	
wcscat( )	*wcscat*( )	
wcsncat( )	*wcsncat*( )	
wcscmp( )	*wcscmp*( )	
wcscoll( )	*wcscoll*( )	*m_wcscoll*( )
wcsncmp( )	*wcsncmp*( )	
wcsxfrm( )	*wcsxfrm*( )	*m_wcsxfrm*( )
wcschr( )	*wcschr*( )	
wcscspn( )	*wcscspn*( )	*m_wcscspn*( )
wcspbrk( )	*wcspbrk*( )	*m_wcspbrk*( )
wcsrchr( )	*wcsrchr*( )	
wcsspn( )	*wcsspn*( )	*m_wcsspn*( )
wcsstr( )		
wcstok( )	*wcstok*( )	*m_wcstok*( )
wcslen( )	*wcslen*( )	
wmemchr( )		
wmemcmp( )		
wmemcpy( )		
wmemmove( )		
wmemset( )		
wcsftime( )	*wcsftime*( )	*m_wcsftime*( )
btowc( )		
wctob( )		
mbsinit( )		
mbrlen( )#		
mbrtowc( )#		
wcrtomb( )#		
mbsrtowcs( )#		
wcsrtombs( )#		

MSE functions marked with a dagger (†) in the above list are defined in the **Distributed Internationalisation Services** snapshot to support multi-locale operation when the **mbstate_t** object associated with the stream has been set by means of *m_fattr*( ) and has a valid locale object associated with it. Otherwise, these functions behave as defined in the MSE.

Functions marked with a hash (#) are similarly defined; that is, MSE behaviour results if the **mbstate_t** object passed as an argument refers to the global locale, multi-locale operation results if the **mbstate_t** object refers to a valid locale object.

Glossary

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

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

coded character
A code value encoded as one or more objects of type **char** that corresponds to a member of the codeset of the locale.

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

code element
Refers to a character encoded as either a wide-character code (of type **wchar_t**) or a *coded character* (of type **char***).

code element string
A contiguous sequence of *code element*s all having the same type and terminated by and including the first null *code element*. A pointer to a *code element string* is a pointer to its initial (lowest addressed) *code element*. The length of a *code element string* is the number of *code element* objects preceding the null *code element*.

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.

codeset
The result of applying rules that map a numeric code value to each element of a character set. An element of a character set may be related to more than one numeric code value but the reverse is not true. However, for state-dependent encodings the relationship between numeric code values to elements of a character set may be further controlled by state information. The character set may contain fewer elements than the total number of possible numeric code values; that is, some code values may be unassigned. The mapping of a unique numeric value to each character in a particular character set.

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

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.

network locale specification
The abstraction for representing the name of a particular locale that is known as a network object. On a host system, a network locale specification is of type **LocaleSpec**. A network locale specification cane be either a *string network locale specification* or a *token network locale specification*.

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 coded character
A coded character with code value 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** on page 48.

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 ! # % ˆ & * ( ) _ + - = { } [ ]
: " ˜ ; ’ ‘ < > ? , . | \ / @ $
```

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.

string network locale specification
A character string of type **LocaleNetString** that unambiguously represents the contents of any locale across the network. The string network locale specification of a locale is invariant across the network and is encoded using the ISO 646 International Reference Version (IRV) codeset.

token network locale specification
A shorthand way of identifying a string network locale specification. A token network locale specification is of type **LocaleNetToken**. Not every possible locale has a token network locale

specification allocated. A token network locale specification for a locale that has been allocated a token, is invariant across the network.

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

basic multilingual plane 47
canonical form 47
character 47
character set 47
character string 47
classification function 17
code element 10, 47
code element string 10, 47
coded character 10, 47
coded character set (codeset) 47
coded character string 10, 47
codeset 28, 47
combining character 28, 47
complex text **28**
  codesets 28
  composite sequence 29
complex-text languages 28
composite sequence 28-29, 48
composite sequence function 32
context-sensistive text 31
control character 48
data type 43
date formatting function 22
diacritic 28, 48
directional text 31
distributed internationalisation 35
empty string 48
extended wide-character conversion function 25
file code 48
formatted I/O function 24
function 44
global locale **3**
  layered software 6
  object-oriented software 5
  stateful encodings 7
  summary 7
  threaded software 6
graphic character 48
internationalisation 48
introduction 1
layered software 6
locale 48
  definition 3
locale information function 15
locale management function 13
locale naming **38**
locale registration 42
LocaleSpec function 39
localisation 48
monetary formatting function 22
MSE change **43**
  data types 43
  functions 44
multi-locale function
  classification 17
  date formatting 22
  extended wide-character conversion 25
  formatted I/O 24
  locale information 15
  locale management 13
  monetary formatting 22
  number conversion 23
  string comparison 21
  string searching 20
  text parsing 23
  text scanning 23
  time formatting 22
  transliteration 19
multi-locale model **9**
  definitions 10
  program flow 11
  terms 10
NetSpec function 41
network locale specification 38, 48
  string 38
  token 39
non-spacing characters 48
null byte 48
null coded character 10, 49
null pointer 49
null string 49
number conversion function 23
object-oriented software 5
octet 49
portable character set 49
portable filename character set 49
process code 49
script 49
setlocale( ) 4
stateful encodings 7
string 49
string comparison function 21

string network locale specification 49
string searching function 20
text parsing function..23
text scanning function 23
threaded software...6
time formatting function...22
token network locale specification 49
transliteration function..19
wide-character code...50
wide-character string...50