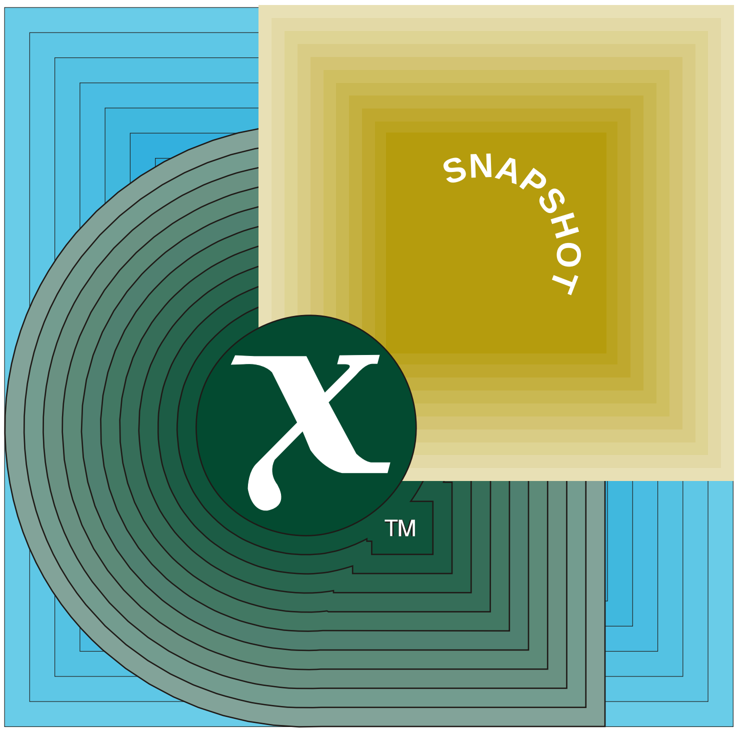
Snapshot

Distributed Internationalization Services Version 2



THE Open GROUP

In conjunction with

UniForum

X/Open Snapshot

Distributed Internationalisation Services, Version 2

The Open Group

X/Open Snapshot

Distributed Internationalisation Services, Version 2

ISBN: 1-85912-033-4
Document Number: S308

Published in the U.K. by The Open Group, December 1994.

Any comments relating to the material contained in this document may be submitted to:

The Open Group
Apex Plaza
Forbury Road
Reading
Berkshire, RG1 1AX
United Kingdom

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Examples

Preface

This Document

This document is a Snapshot (see above).

The current model for producing internationalised software, as presented in issues of the X/Open Portability Guide (XPG) up to and including Issue 4, was developed at a time when the norm was stand-alone systems with terminals. For such environments, the current global internationalisation model based on *setlocale*( ) is a reasonable solution, and will continue to be a reasonable solution for software intended for these environments.

However, the current internationalisation model has limitations when applied to software designed:

- to be distributed
- to work in heterogeneous networks
- to interact simultaneously with multiple users, each of which uses different language customs and conventions
- to be part of a layered solution.

This was exposed during the X Consortium's attempt to internationalise the release of X11R5.

This document represents the joint recommendation of the X/Open and UniForum Joint Internationalisation Group to address limitations of the current internationalisation model.

Structure

This document is structured as follows:

- Chapter 1 is an introduction explaining the background to this snapshot.
- Chapter 2 summarises problem statements in the areas of locale registry, multi-locale environments, multi-lingual applications, distributed processing and text objects; it then proposes solutions to the problems identified.
- Chapter 3 describes the syntax and semantics of the naming scheme required to identify a locale across a heterogeneous network.
- Chapter 4 describes functions for the creation and management of non-global locale objects, and functions intended to overcome perceived drawbacks with existing methods of internationalisation.
- Chapter 5 provides a specification for the header file.
- Chapter 6 provides detailed specifications for the functions introduced in Chapter 3 and Chapter 4.
- Appendix A explains the scope of the X/Open Locale Registry.
- Appendix B describes other proposals for locale management and gives the rationale for their rejection.

A glossary and index are provided.

Intended Audience

This document is intended for system programmers and application programmers working on distributed internationalised systems and applications.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for options to commands, filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — command operands, command option-arguments or variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — utility names
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( ); names without parentheses are C external variables, C function family names, utility names, command operands or command option-arguments.
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C **#define** construct.
- The notation [EABCD] is used to identify an error value or coded return 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.
- Ranges of values are indicated with parentheses or brackets as follows:
  - — (a,b) means the range of all values from a to b, including neither a nor b
  - — [a,b] means the range of all values from a to b, including a and b
  - — [a,b) means the range of all values from a to b, including a, but not b
  - — (a,b] means the range of all values from a to b, including b, but not a.
- Hexadecimal numbers are denoted by a prefix of 0x or 0X.

Trade Marks

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

Hewlett-Packard®, HP®, HP-UX®, and Openview® are registered trademarks of Hewlett-Packard Company.

IBM® is a registered trademark of International Business Machines Corporation.

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

Acknowledgements

X/Open gratefully acknowledges the collaboration with and contributions from UniForum.

Many members of the X/Open and UniForum Joint Internationalisation Group have contributed to this document by writing or reviewing drafts. In particular, thanks are due to Frank Rojas of IBM for his work on this version.

Special thanks are due to Tom McFarland of HP for work on earlier drafts and continued support of this effort. Also Tim Greenwood of Digital for his comments that help align the DIS with the MSE functions.

Finally, thanks to Judy Chen of HP for some of the most thorough reviews of this and other XoJIG specifications.

Referenced Documents

The following standards are referenced in this snapshot:

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

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

UTF-8
: CAE Specification, April 1995, File System Safe UCS Transformation Format (UTF-8) (ISBN: 1-85912-082-2, C501), published by The Open Group.

ISO 639
: ISO 639: 1988, Codes for the Representation of Names of Languages, Bilingual edition.

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

ISO 3166
: ISO 3166: 1988, Codes for the Representation of Names of Countries, Bilingual edition.

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

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

ISO 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 C
: ISO / IEC 9899: 1990: Programming Languages — C, including:
: Amendment 1: 1995 (E), C Integrity (Multibyte Support Extensions (MSE) for ISO C).

ISO POSIX-1
: ISO / IEC 9945-1: 1996, Information Technology — Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) [C Language] (identical to ANSI / IEEE Std 1003.1-1996). Incorporating ANSI / IEEE Stds 1003.1-1990, 1003.1b-1993, 1003.1c-1995 and 1003.1i-1995.

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 as amended by IEEE Std 1003.2a-1992).

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

POSIX.1c
IEEE Std 1003.1c / D?, Threads Extension for Portable Operating Systems, <date>.

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

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

Issue 3
XSI Internationalisation, in Chapters 2 to 8 inclusive of: X / Open Specification, Issue 3, 1988, 1989, February 1992, Supplementary Definitions, Issue 3 (ISBN: 1-87263-38-3, C213); this specification was formerly X / Open Portability Guide, December 1988, Volume 3, (ISBN: 0-13-685850-3, XO / XPG / 89 / 003).

and

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

Issue 4
The **XSH, Issue 4** specification (see below).

XBD, Issue 4
CAE Specification, July 1992, System Interface Definitions, Issue 4 (ISBN: 1-872630-46-4, C204), published by The Open Group.

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

X11R4 X File Formats
CAE Specification, August 1991, Window Management: X Window System File Formats and Application Conventions (ISBN: 1-872630-15-4, C170), published by The Open Group.

Version 1 of this document
X / Open Snapshot, November 1992, Distributed Internationalisation Services (ISBN: 1-872630-75-8 S213).

Federated Naming
Preliminary Specification, August 1994, Federated Naming: The XFN Specification, (ISBN: 1-85912-458-8, P403), published by The Open Group.

Chapter 1

Introduction

This chapter explains the background to the development of internationalisation services for distributed systems, and defines the scope of this document.

1.1 Background

The current internationalisation model was first published in Issue 2 of the X/Open Portability Guide (1987). Since then, it has undergone changes through subsequent issues, as follows:

Issue 3 (1988)

For the first time internationalisation facilities were defined as mandatory in the **X/Open System Interface**. Issue 3 was also fully aligned with the POSIX.1 standard and ANSI C, except in the area of multi-byte codeset operation and *localeconv*( ).

Issue 3 included all the facilities presented in Issue 2, modified for alignment with the above standards, plus:

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

Issue 4 (1992)

The main changes in this issue are support for Asian languages and a more comprehensive definition of the internationalised utility environment. Interfaces are also added that provide codeset conversion facilities at the program level.

In summary, Issue 4 includes all the facilities published in Issue 3 plus:

- Worldwide Portability Interfaces, which enable applications to work with either single-byte or multi-byte codesets
- extended support of the internationalised utility environment
- additional system interfaces for date and time conversion (*strptime*( )), monetary value conversion (*strfmon*( )) and codeset conversion (*iconv**( ))
- full conformance to the ISO C standard
- the *localedef* and *locale* utilities.

A similar global internationalisation model has also been adopted in the referenced ISO POSIX-1 standard and the referenced ISO C standard, but the basic foundation of the model remains unchanged; that is, one application equals one language, territory and codeset per instantiation. This is achieved through use of an opaque application global structure — one per process. The model was deemed adequate for current software technology when it was developed, and it has been widely applied by many of the major software and hardware vendors.

In 1990, the X Consortium members began pressuring for the X Window System to be internationalised. Until then, it was designed only to work with a single codeset (as defined in the ISO 8859-1 standard). To work with other codesets, or languages other than those used in the Americas and Western Europe, it was left to the application to handle font management, produce characters from the keyboard and to break strings into segments for rendering (one segment per constituent character set in the codeset of the locale). The X Consortium's mltalk work group began a two year effort to define, specify and provide a sample implementation of X that was in line with the current internationalisation model based on *setlocale*( ).

During the mltalk work group's investigations, it became apparent that while the internationalisation model based on *setlocale*( ) worked well for terminal-based applications, it became cumbersome, and even obstructive, when attempting to apply it to:

- object-oriented software
- layered software
- distributed software
- threaded software
- multi-user software
- advanced text handling (contextual or bidirectional) software.

The X Consortium applied what they could use of the *setlocale*( ) model. Several X/Open members who were active in the development of the X11 release 5 specification presented to the X/Open-UniForum Joint Internationalisation Group (JIG) the difficulties encountered in the release 5 procedure.

The X Window System is just one example of many types of software that needs additional distributed internationalisation services to be portable across various vendors' systems. Since the initial work on this document, other groups have identified needs for these services. Recently, the WG21 C++ group started working on defining a set of *locale objects* that provide similar services to those defined in this document.

The first version of this document (see **Referenced Documents** on page xi) was the result of generating solutions to the problem areas identified above. This document is a revision based on industry feedback. Chapter 2 offers a more detailed description of the problem statements.

1.2 Scope

After reviewing the problems outlined above, the X/Open-UniForum JIG felt that these were pervasive and important enough to require the definition of new services. In short, the existing **XSH, Issue 4** specification internationalisation functions do not offer a sufficiently rich environment for the development of new internationalised technologies expected to work in distributed, threaded and object-oriented environments.

This document offers a method of announcing locale. It also provides functions that satisfy multi-locale, multi-threading and multi-node processing.

This document does not provide a complete solution to all of the problems described in Chapter 2, but does establish the foundation for distributed internationalisation applications. This document is one part of a complete solution. Successful application of the non-global locale model for such software depends on further work to resolve issues such as those described in Section 1.2.3 on page 5.

1.2.1 Areas Addressed

The proposed solutions are additions to the current internationalisation model as follows:

- A locale syntax — this is required for identifying a given locale across a network.
- A locale registry — this is required to supply semantics for certain well agreed upon locales. Using the proposed new syntax, a name for each registered locale is defined ensuring that language-sensitive operations using that locale on system A obtain the same results when performing those operations on system B (where systems A and B may be from different vendors). Refer to Chapter 3 for more information.
- Multi-locale and advanced text capability — these provide:

  — functions that operate with a locale object or handle

  A new set of APIs that operate on a per locale handle is defined. The new functions are based on making it easier for software to manage and use multiple locales without impacting other software layers. Further, these APIs also resolve the complaints against the global nature of the existing **XSH, Issue 4** specification or ISO C locale model voiced by developers of object-oriented programs (global data is the antithesis of the object-oriented programming paradigm).

  — functions that support advanced text properties

  In the new APIs, there is provision for the manipulation of advanced text properties, meeting the requirements of both context-sensitive rendering and directionality. All locale-based capability only processes mixed directional text that is in logical order (not in its final presentation or visual order.)

  — support for importing and exporting stateful encodings

  An abstraction called a *text context object* (**mbstate_t**) is defined to allow text processing that requires a context to be maintained across various function calls. The text context object is primarily intended for the import and export of stateful data. Yet the text context object may be extended for other specific cases where a context is needed, for example, tokenising a string.

  Refer to Chapter 3 for more information on all aspects of multi-locale and advanced text capability.

- Clarification of existing **XSH, Issue 4** specification capability:
  - — The existing *setlocale*( ) model is not designed to support stateful encodings, despite the implications of functions such as *mblen*( ).
  - — A definition of *setlocale*( ) is provided that is suitable for a threaded environment as defined in the POSIX.1c draft standard.

1.2.2 Areas Not Addressed

This document does not:

- Provide data tagging
  - — This document does not define specific data tagging encoding nor functions for handling data tags. X/Open expects that higher-level subsystems will define the data tagging handling that best fits their environments. This document makes no policy on data tagging, but instead provides the functions that can operate on individual tagged data segments on a per locale basis, independent of a global locale state.
  - — This document does not prohibit data tagging in the future.
- Obsolete existing capability — the non-global locale model does not obsolete the global locale model; instead it provides an alternative method of internationalising applications. The trade-offs in the use of each model, together with the requirement of the application, determine the software developer's choice. Future demand by software developers will determine if either or both are required in the future.
- Provide a total solution for multi-lingual applications — the non-global locale model does not provide a complete tool box for the developer of multi-lingual applications. However, it does address one of the key problems for such software today.

This document does not provide new programmatic interfaces for message catalogue use. It was felt that the existing global locale mechanism was more than sufficient. The locale affects messaging only at message catalogue open time, and then only to determine which file to open. It seemed wasteful to create a new catalogue open routine when the existing routine could be used.

For example, if an application needs to open an attribute object-specific message catalogue, the following sequence of functions can be used:

```
char * orig loc; /* to hold the original LC_MESSAGES locale value */
nl catd att catd; /* attribute object spec. catalogue descriptor */
orig loc = setlocale(LC_MESSAGES, NULL);
(void) setlocale(LC_MESSAGES, m_setlocale(AttrObject, LC_MESSAGES, NULL));
att catd = catopen("app name", NL_CAT_MESSAGES);
(void) setlocale(LC_MESSAGES, orig loc);
```

The host locale name for the LC_MESSAGES category of an attribute object can be retrieved with *m_setlocale*(AttrObject, LC_MESSAGES, NULL). In the above code, it is used to initialise the global locale's LC_MESSAGES category so that *catopen* will find the attribute object-specific message catalogue.

There is potential that the global locale could be changed by another object between the *setlocale*( ) function and the *catopen*( ) function. But since the only part of the locale being modified is LC_MESSAGES, since the window of time in which this would actually be a problem is very small, and since the only procedure affected by the LC_MESSAGES category is called only once or twice in a typical application, the risk was deemed too small to justify creation of yet another new API.

1.2.3 Issues

When a language-sensitive remote procedure is executed, the results of that procedure on system A should be the same as on system B. The important questions are:

- How does the application-side system convey the desired locale to the remote system?
- How can the application be sure that the remote system supports the desired locale?

At some point in the future, the issue of which locale is associated with which data must be addressed. Currently, it is left to the software developer to design and implement his own mechanism.

1.3 Terminology

The following terms are used in this specification:

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

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

legacy
Certain features are *legacy*, which means that they are being retained for compatibility with older applications, but have limitations which make them inappropriate for developing portable applications. New applications should use alternative means of obtaining equivalent functionality. Legacy features are marked **LEGACY**.

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

must
This describes a requirement on the application or user.

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

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

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

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

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

Chapter 2

Overview of Problem Areas

This chapter is a summary of the problem statements generated by the X/Open-UniForum JIG while analysing the complaints from the X Consortium. The proposals made to overcome the problems are covered in the following chapters.

2.1 Locale Registry

The ISO C standard and the ISO POSIX-1 standard define a locale where a variety of culture-dependent data is defined, including language-dependent specifications for date and time, collating sequences, and numeric and monetary conventions. Many standard utilities in the ISO POSIX-2 standard use these specifications to provide localised versions of the utility.

Currently, there is no general way to specify portably which locale a user wants to use with these standard utilities. Only one, non-internationalised locale is standard: the C or POSIX locale.

A registry that uniquely names all locales and distributes such information to users would be useful.

Further, in distributed software, such as that using Remote Procedure Calls (RPCs), it is imperative that the semantics of a given locale on one system are identical to the same locale on a different system.

2.2 Multi-Locale Environments

How should a system behave in an environment where multiple locales, in particular codesets, are supported within a single system or network? What support is provided in the system to detect and correct collisions in names of system resources due to codeset overlapping? Specifically, consideration has to be given to how these names are used in a networking environment as well as in a single host environment.

Consider the following environments:

- A window manager manages multiple clients, each with its own locale. When the window manager displays a title for a client, it should use the locale of the client, not the locale of the window manager.
- Print servers, which may get requests from various clients, each run their own locale. Any error messages sent to the client should be in the locale of the client, not in the locale of the print server.
- A database application needs to associate the date format with the date of each item in the inventory.

The simplicity of the current global locale model makes it easy to write internationalised programs. At the same time it makes it difficult to write applications that deal with many different languages and territories simultaneously. Examples of such programs include:

- a spreadsheet program that uses multiple currencies and date formats
- ledger programs for international companies
- database programs merging and processing databases from different languages or codesets
- window managers

  These are programs that control and label the windows of all applications displayed on a workstation.
- Xt library

  This is the portion of the library that creates the resource database for an application, merging databases from several systems and sources, each of which could be created in a different language and codeset.
- window-based process-control application

  This is a single application that interacts with many different users. Each user wants to interact in his own language. Each user expects data to be processed (for example, number formats, time and date, sorted lists) in accordance with local rules.
- a word processing application for multi-language texts (such as translated documents, international standards and annotated literature), with needs for language-sensitive hyphenation, justification, and so on.

Such multi-locale applications need to be able to call functions that provide the current internationalisation capability for data consisting of an arbitrary mixture of languages and codesets. The procedure for doing this must not be so complex that programmers do not use it. Ideally, there should be one standard way to identify the language or codeset of a data segment instead of forcing applications to invent their own methods.

2.3 Distributed Processing

Consider two cases of remote procedure call (RPC) dealing with international text in a heterogeneous network. The simple case (single thread, single locale, global locale state) allows us to propose an approach for handling international text. The complicated case (multiple threads, multiple locale, global locale state) illustrates RPC problems that cannot be resolved with the current locale model.

Independent processes running in a heterogeneous network cannot communicate with each other in a multi-lingual environment due to lack of coherent locale support in performing remote procedure calls.

2.3.1 Single Thread, Single Locale, Global Locale State

The client-server paradigm is used for general Operating System (OS) discussion, not only for X Windows. The basic approach proposed here applies to general operating system RPC usage:

1. Client announces to the server the locale it is running.
2. Server accepts the locale or negotiates some other common locale. If nothing in common, server rejects the connection.

   This requires a standardisation of locales, perhaps an ISO, POSIX or X registry.
3. If the locale can be supported by the client-server pair, the client-server pair is compatible.
4. Compatible client-server pairs then negotiate on encoding and data type.

   Encoding:

   — compound text encoding (or some equivalent) as least common denominator

   — other recognised encoding (for example, ISO 8859-n, ISO 10646 as UCS-2 or UCS-4).

   Clients and servers from the same vendor may not need to use compound text; it may be possible to agree on a particular encoding suitable for both client and server.

   Data type:

   — multi-byte format as least common denominator

   — **wchar_t** type, with a commonly agreed number of bytes.

   Encoding and data types beyond the least common denominators are used for optimisation purposes, to avoid the need to convert to multi-byte compound text for all cases.

The same basic approach can be applied to X Windows client-server connections. Using Inter-Client Communications Conventions (ICCCM),[1] X text exchange is simpler. However, the need for agreement of standard locales in a heterogeneous network still exists. Therefore steps 1 and 2 still need to be done.

1. The ICCCM specification is in Chapter 2 of the X File Formats specification.

2.3.2 Multiple Threads, Multiple Locales, Global Locale State

For a distributed application that uses multiple threads (for example, pthreads as in P1003.1c) even on a uni-processor system there are some currently unresolved issues with the global locale model. The problems encountered by remote procedure calls also exist in local function calls. These issues fall into three categories.

Global State of the Locale

Explicit Argument (RPC)

For a function that is to be invoked remotely, the calling interface passed between the client and server is an interchange definition of the arguments to the client stub (and also possibly the call context handle). Since the current global locale is not an explicit argument to any functions, any interface definition language that generates the client stub does not have enough information on whether or not the locale should act as an argument to any or all functions.

Thread Interaction (Threads)

In a threaded environment, two or more threads may be in execution phase at any given time. If either of these threads affects the default locale, any locale-sensitive operations in the other thread are affected:

- Global state cannot be maintained by saving and restoring the locale without blocking other threads.
- Locale-sensitive functions cannot be guaranteed to take place in a given locale without blocking all other threads.

Locale Synchronisation

Even if it were possible to specify a locale as an additional argument to locale-sensitive functions, it cannot be guaranteed that if a client passes some locale identifier to a server, that the identifier can be used to reconstruct the original locale. Some type of locale agreement needs to take place.

The *setlocale*( ) syntax allows the current locale to be interrogated and its name to be returned. There is no guarantee that this locale could be replicated on the RPC server correctly. Compare this to the situation where the locale could be identified as a handle to some locale structure, which could be interchanged as an RPC object.

Type wchar_t in a Distributed Environment

Because the representation of type **wchar_t** is implementation defined, and the mapping of multi-byte characters to wide-character codes is locale sensitive, a function with a type **wchar_t** interface is not easily distributed. For example, performing an RPC call on a machine with a 4-octet type **wchar_t** to a machine that represents **wchar_t** as two octets, requires cooperation at the application programming level for data representation, and at the system and locale level for the multi-byte character to wide-character code mapping.

2.4 Advanced Text Handling and Encoding

The **XSH, Issue 4** specification perpetuates the ISO and ANSI paradigm of presenting basic text entities that are closely associated with a glyph or display cell. This causes problems in supporting languages and codesets that use combining characters. For example, 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. This requires functions that can handle characters represented by composite sequences just as the **XSH, Issue 4** specification was enhanced to handle multi-byte characters. These functions must handle any composite sequence as a single entity for classification, manipulation, passing through the I/O system, and so on.

Another complication in this area concerns text directionality. Certain languages require that text objects be processed left-to-right or right-to-left, depending on the character class of a 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.

Chapter 3

Locale Specification for Distributed Environments

This chapter describes the syntax and semantics of the naming scheme required to identify a locale across a heterogeneous network.

3.1 Purpose

The purpose of defining a naming scheme is to allow locales to be replicated across a heterogeneous network. For any form of distributed processing, and RPC in particular, the distributed code must, by default, work in an environment replicated from the client. To replicate this internationalised environment, the locale on the client must be unambiguously identified so that it can be replicated in the server.

3.2 Terminology

The following new terms are used in this section:

host locale string
locale object
LocaleSpec (data type)
network locale specification (in either token or string form)
LocaleNetToken (data type)
LocaleNetString (data type)

The abstraction for representing the name of a particular locale on a host system is called a *host locale string*.

The abstraction for representing the contents of a particular locale that is known as a host system object is called a *locale object*. On a host system, a locale object is of type **AttrObject**.

The abstraction for representing the name of a particular locale that is known as a network object is called a *network locale specification*. On a host system, a network locale specification is of type **LocaleSpec**. On a network, a network locale specification may be a *token network locale specification* (type **LocaleNetToken**) or a *string network locale specification* (type **LocaleNetString**).

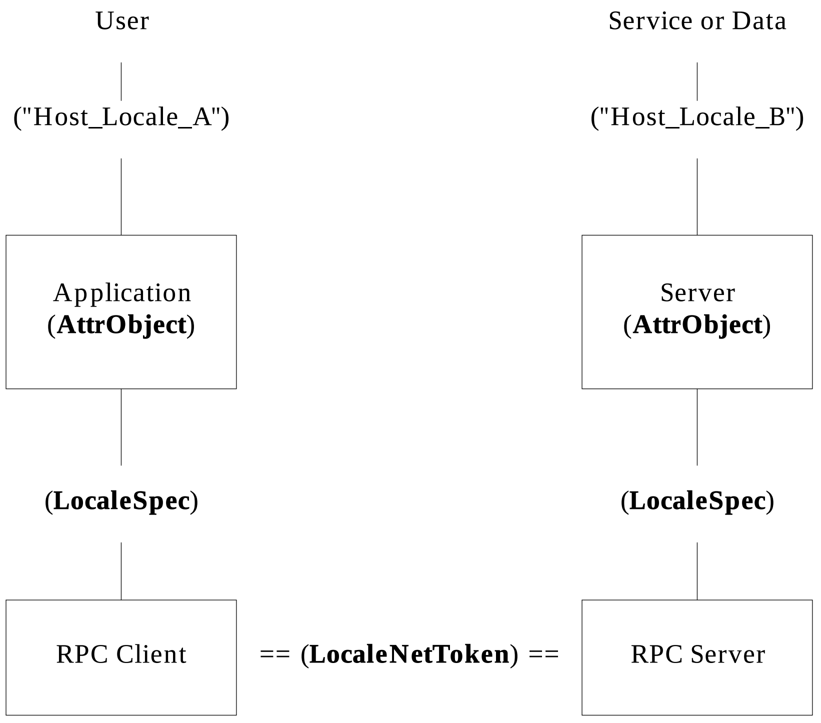
3.3 Locale Usage for Host and Network

The string currently returned as the result of:

```
setlocale(LC_ALL, NULL);
```

unambiguously identifies a given locale within a process, on a given implementation. However, a host locale string is implementation dependent and therefore cannot be interchanged between implementations.

To enable consistent localised behaviour of distributed applications a well known representation is needed for the locales. The term *network locale specification* refers to an object that can be exchanged across the network and is guaranteed to represent a set of well known locale categories. An example of the expected flow of information is as follows:



In the figure, the host locales are represented as strings that may be implementation dependent: "Host_Locale_A" and "Host_Locale_B". A distributed application can map the host locale string into a **LocaleSpec** which is used to map the host locale string into a network locale specification. The **LocaleSpec** provides an opaque representation of the network locale specification for efficient communication with the host. Because the RPC run-time services need to communicate with the locale specification, functions are provided to map the opaque **LocaleSpec** into a well known format for communication (a **LocaleNetToken** or a **LocaleNetString**). More information about **LocaleSpec**, **LocaleNetToken**, **LocaleNetString** and other programming data types are provided in Section 4.3 on page 25.

The string network locale specification provides the name for each category that exists within the locale. Not all categories from the standard category list must be present in the network name. The string network locale specification does not specify the LC_ALL category but instead calls out each specific category, since the specification of LC_ALL is itself ambiguous across implementations that may support optional categories.

Within an implementation, certain extended categories may be supported in a locale. The support of such extended categories on a network-wide basis is optional. The string network locale specification explicitly lists each optional category, denoting it as such by the use of the prefix OPT_, followed by the implementation-defined name of the category.

3.4 String Network Locale Specification Syntax

Any given locale may be comprised of one or more categories, some of which are enumerated and specified in various standards or industry specifications (ANSI C, ISO C, POSIX, XPG). A given implementation may include optional categories over and above those specified in the standards indicated.

To identify a locale, each of the standard categories and their values must be enumerated, along with the names and values of each vendor-specific category supported in the locale. The specification takes the form of a list of keyword-value pairs for each of the categories.

In all cases the string is composed of characters from the ISO/IEC 646 International Reference Version (IRV): 1990 codeset, with the exception of codepoints in the range 0 to 32 (decimal) and the codepoint 127 (decimal), which are reserved for syntax specification.

The following symbols are used in the proposed grammar for string network locale specifications given below:

Character	**Value**	**Description**
,	44	Comma
-	45	Hyphen
/	47	Solidus (Slash)
;	59	Semi-colon
=	61	Equals sign
_	95	Low line (Underscore)

An ellipsis symbol (...) represents missing enumerated values of that type, and the string **opt_** precedes optional segments.

```
charnum_list          : charnum_list charnum
                      | charnum
                      ;
charnum               :'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'|'d'|'d'|'e'|'f'|'g'|'h'|'i'|'j'
                      |'k'|'l'|'m'|'n'|'o'|'p'|'q'|'r'|'s'|'t'
                      |'u'|'v'|'w'|'x'|'y'|'z'|'-'|'_'
                      | number
                      ;
number                : number digit
                      | digit
                      ;
digit                 :'0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'
                      ;
network_locale_string : std_cat_specs opt_cat_specs
                      | std_cat_specs
                      ;
std_cat_specs         : std_cat_specs std_cat_spec
                      | std_cat_spec
                      ;
std_cat_spec          : ctype_spec | collate_spec | messages_spec
                      | monetary_spec | numeric_spec | time_spec
                      ;
```

```
ctype_spec            : ctype_keyword full_category_spec delimiter
                      ;
collate_spec          : collate_keyword full_category_spec delimiter
                      ;
messages_spec         : messages_keyword full_category_spec delimiter
                      ;
monetary_spec         : monetary_keyword full_category_spec delimiter
                      ;
numeric_spec          : numeric_keyword full_category_spec delimiter
                      ;
time_spec             : time_keyword full_category_spec delimiter
                      ;
delimiter             : '/'
                      ;
ctype_keyword         : 'CTYPE='
                      ;
collate_keyword       : 'COLLATE='
                      ;
messages_keyword      : 'MESSAGES='
                      ;
monetary_keyword      : 'MONETARY='
                      ;
numeric_keyword       : 'NUMERIC='
                      ;
time_keyword          : 'TIME='
                      ;
full_category_spec    : registry_spec    ';'
                        name_spec        ';'
                        ver_spec         ';'
                        encoding_spec    ';'
                      ;
registry_spec         : charnum_list
                      ;
name_spec             : charnum_list
                      ;
encoding_spec         : charnum_list
                      | XFN_encoding
                      ;
XFN_encoding          : 'XFN_'hexnumber
                      ;
hexnumber             : hexnumber digit
                      | hexnumber charhex
                      | digit
                      | charhex
                      ;
charhex               : 'a' | 'b' | 'c' | 'd' | 'e' | 'f'
                      | 'A' | 'B' | 'C' | 'D' | 'E' | 'F'
                      ;
ver_spec              : major_ver '_' minor_ver
                      ;
major_ver             : number
                      ;
```

```
minor_ver               : number
                        ;
opt_cat_specs           : opt_cat_specs  opt_cat_spec
                        | opt_cat_spec
                        ;
opt_cat_spec            : opt_cat_keyword opt_cat_value delimiter
                        ;
opt_cat_keyword         : 'OPT_' charnum_list '='
                        ;
opt_cat_value           : charnum_list '_' charnum_list
                        | charnum_list
                        ;
```

From the description above it is evident that there are several character string literal entries whose values must be standardised for network naming to succeed. These are:

- registry_spec
- name_spec
- encoding_spec.

The hexnumber of an XFN_encoding should identify a registered encoding defined in the **Federated Naming** specification.

Notes: While permitted by the above grammar, multiple codesets should not be mixed within a single network locale specification. The syntax that allows this (that is, `full_category_spec`) is retained for compatibility with the **XSH, Issue 4** specification.

3.4.1 Examples

Note: Line breaks have been inserted at the end of each category for readability. They should not appear in a valid string network locale specification.

Example 3-1 American English Locale

This is as stipulated by ANSI using the ISO 8859-1 (Latin alphabet No. 1) codeset. The XFN-encoding used is defined in the **Federated Naming** specification:

```
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;/
```

Example 3-2 ISO Japanese Locale

This uses AJEC (Japanese EUC). The XFN-encoding used is defined in the **Federated Naming** specification:

```
CTYPE=ISO;ja_JP;01_00;XFN-00030010;/
COLLATE=ISO;ja_JP;01_00;XFN-00030010;/
MESSAGES=ISO;ja_JP;01_00;XFN-00030010;/
MONETARY=ISO;ja_JP;01_00;XFN-00030010;/
NUMERIC=ISO;ja_JP;01_00;XFN-00030010;/
TIME=ISO;ja_JP;01_00;XFN-00030010;/
```

Example 3-3 French Canadian Locale

This is a mixed locale created in an IBM environment with an OSF North American time category. The XFN-encoding used is defined in the **Federated Naming** specification:

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

Example 3-4 HP German Locale

This is created with only the CTYPE category:

```
CTYPE=HP;de_DE;01_00;XFN-number;/
```

Example 3-5 X/Open de_DE Registered Locale

This is as stipulated by X/Open's Locale Registry for a German in Germany locale using the ISO 8859-1 standard codeset. The XFN-encoding used is defined in the **Federated Naming** specification:

```
CTYPE=XOPEN;de_DE;01_00;XFN-001001;/
COLLATE=XOPEN;de_DE;01_00;XFN-001001;/
MESSAGES=XOPEN;de_DE;01_00;XFN-001001;/
MONETARY=XOPEN;de_DE;01_00;XFN-001001;/
NUMERIC=XOPEN;de_DE;01_00;XFN-001001;/
TIME=XOPEN;de_DE;01_00;XFN-001001;/
```

3.5 Token Network Locale Specifications

The syntax defined in Section 3.4 on page 16 can identify any given locale at the expense of the length of the specification itself (some 200+ bytes for a standard American English locale). In some environments the length of this specification is not an issue. However in some cases it may be, in particular for data tagging.

This proposal includes the notion of a network syntax that includes certain predefined shorthand values, referred to as token network locale specifications. It is proposed that each token be an unsigned integer value, representable within four octets, in the format described below. In particular, a token value could possibly form the value field of an object identifier within ASN.1 syntax, to describe a locale for data tagging (for example, within an ODA document the tag could be at a text item level; within RPC it could be at the session connect level).

The format of the token network locale specification is an unsigned integer of four octets. A token for a given locale implies that all the standard categories have the same locale name.

The two most significant octets of the four-octet group represents the registration authority. OSF is maintaining a codeset registry. National and international standards bodies, companies, consortia, and so on, who wish to use network locale specification tokens are allocated unique identifiers in the OSF registry. A block of values is reserved for private use between consenting systems; the block of values will never be allocated by OSF.

This scheme allows some 65,535 registration authorities, with each registration authority having a registration space of 65,535 individual tokens, each token representing a particular national profile.

The Locale Number defined in the least significant two octets is a number assigned by the registration authority to a particular national profile (or in the case of a manufacturer, to a manufacturer-supported locale). No assumptions should be made about the value ranges of the locale numbers assigned by any national or international body, unless explicitly stated to this effect by the appropriate body.

Within a program a string network locale specification is described by the type **LocaleNetString** (which must be a type **char***). The 32-bit token network locale specification is described by the type **LocaleNetToken**. Both forms are defined because different protocols may require different forms. The tokens are intended for low-level communication layers, not for application programs. Both types of network locale specification can be handled within the object of type **LocaleSpec**. The type **LocaleSpec** is introduced to make it convenient for programs to represent a network locale specification as a single form. Again, only low-level services need be concerned with the type **LocaleNetString** and **LocaleNetToken**.

Refer to Section 4.4 on page 28 for information about mapping between these types.

3.6 Registration

X/Open has a registry for the storage of locales. Where possible, these locales are taken from existing national profiles for the appropriate country (see Appendix A).

Chapter 4

Multi-locale Support

This chapter provides an overview of the functions defined to support multi-locale programs in a distributed environment. These features address the major drawback with the *setlocale*( ) function and the ISO 3166 standard dependency on a single locale per program on a single system, and help promote international portability of C programs.

Objectives

The objectives of the multi-locale support functions are to:

- define the base multi-locale processing needed for today's distributed and object-oriented internationalised environments; the processing must:
  - — satisfy internationalised object-based software
  - — satisfy layered internationalised software (for example, libraries)
  - — satisfy programs for multi-node processing in internationalised distributed networks
  - — satisfy multi-threaded internationalised programs
  - — satisfy the multi-locale support requirements encountered in the windowing environment
- address the limitations that reliance on global data places on object-oriented programming paradigms
- co-exist with the global locale functions
- address the problem of stateful encodings
- provide management functions for new objects
- ensure the functions provided can support multi-lingual capability, for example: phrase or word recognition, collation, subsetting of character sets, and so on
- support languages that may have mixed directional characters
- support languages that have composite sequences whose presentation is contextual
- ease transition of applications from the global locale model to the new non-global locale model.

Assumptions

The multi-locale support functions are based on the assumption that the locale registry exists.

The descriptions and terms used in this chapter assume the reader is familiar with several extant standards and specifications, including:

- the ISO C standard
- the MSE standard
- the ISO POSIX-1 standard
- the ISO POSIX-2 standard
- the **XBD, Issue 4** specification
- the **XSH, Issue 4** specification.

4.1 Definitions

In addition to the terms introduced in Section 3.2 on page 13, the following terms are used throughout the rest of this document.

The term *global locale* or *current locale* is used to refer to a particular locale on a host system whose contents (information, data or processing) are visible to an entire process on a single host system.

A *wide-character code* is a code value (a binary coded integer) of an object of type **wchar_t** that corresponds to a member of the codeset of the locale on a host system.

A *null wide-character code* is a wide-character code with code value zero.

A *wide-character string* is a contiguous sequence of wide-character codes terminated by and including the first null wide character. A *pointer to a wide-character string* is a pointer to its initial (lowest addressed) wide-character code. The *length of a wide-character string* is the number of **wchar_t** objects preceding the null wide-character code and the *value of a wide-character string* is the sequence of code values of the contained wide-character codes, in order.

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

A *null coded character* is a coded character with code value zero.

A *coded character string* is a contiguous sequence of coded characters terminated by and including the first null coded character. A *pointer to a coded character string* is a pointer to its initial (lowest addressed) coded character code. The *length of a coded character string* is the number of objects of type **char** (usually bytes) preceding the null coded character code. The *value of a wide-character string* is the sequence of code values of the contained coded characters, in order.

The term *code element* refers to a character encoded as either a wide-character code (**wchar_t**) or a coded character (**char***) that corresponds to a member of the codeset of the locale. A null code element is either a wide character (**wchar_t**) or a coded character (**char***) with code value zero.

The term *code element string* is a contiguous sequence of code elements 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 and the *value of a code element string* is the sequence of code values of the contained code elements, in order.

4.2 Text Model Overview

4.2.1 Basic Text Entity

The multi-locale functions attempt to provide a complete model for the manipulation of generic text entities that can be classified, converted, transferred to and from file store, and so on. The capability of the interfaces is designed to parallel the capabilities that can be performed on **char** types in the C language.

The multi-locale functions extend the ISO, ANSI and the **XSH, Issue 4** specification paradigm of dealing with character entities of specific classes of encodings (single-byte and multi-byte) to the paradigm of dealing with any code element. As such, the multi-locale functions should be viewed as operating on code element strings of any data type. While this document uses the **wchar_t** or **char*** data types to describe the signature of functions, the intent is to describe the functional capability independent of the data type used to encode code elements. Thus, for example using C++, the multi-locale functions could be overloaded to operate on other data types such as an opaque text object or UCS code elements in the future.

4.2.2 Composite Sequences

It is important to note that several of the multi-locale functions are intended to operate on code element strings that may contain composite sequences and as such those functions, for example *m_wcsscanfor*( ), are defined to take arguments that define code element strings rather then a single code element.

For example, representation of the classification unit may be either a precomposed accented character, or a base character followed by one or more non-spacing diacritic characters. In both cases the caller obtains the same results for a classification request, irrespective of the underlying representation of the code element.

More importantly, functions that may potentially be transforming composite sequences from one form to another may have a difference between the input and output code element count. For this reason any classification request is defined to operate on code element strings rather than individual code elements.

4.2.3 Self Announcing Data

The multi-locale functions are aligned with the functions defined in the MSE standard (see **Referenced Documents** on page xi) and are capable of supporting encodings that may have state introducers. In addition, the multi-locale functions serve as the basis for future utilities that may be planning some form of language tagging. The different classes of encodings and how the multi-locale functions address them are described in the following sections.

Character Set Context

There are many encodings that use character set introducers to support mixing multiple character sets within a text data stream. Examples are ASN.1 and ISO 2022, which are in use today. Some implementations claim that character set contexts are sufficient for multi-lingual handling; others claim that this is not sufficient. For example, an encoding may use a Latin-1 (ISO 8859-1) character set introducer, which is not sufficient as a language introducer. On the other hand, a Japanese character set introducer could be viewed as a valid language introducer. In summary, a character set context cannot be used reliably for support of multi-lingual processing. See **Language Context** on page 24 for information on an additional introducer needed to support multi-lingual handling.

While there may be encodings that use character set introducers, any function that operates on code elements is limited to the character set associated with a locale object. Therefore any import or export of such encodings is ultimately limited to the repertoire of the locale object's character set.

Presentation Context

Many encodings use direction introducers (directional controls) as a means to handle scripts that have mixed directional characters. In addition, other encodings have characters whose presentation is said to be context sensitive, for example Arabic scripts and composite sequences defined by FSS-UTF (UTF-8) require presentation services to consider surrounding code elements.

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

For mixed-directional text, all functions based on a locale object are said to operate on a code element string which must be in *logical* order (in the order keyed in). For example, a localised collation function only behaves correctly if the incoming text is in logical order. Any text that is in *visual* order (in the order presented) needs to be *transformed* back into logical order before any collation function can operate on the text.

Given that all text is expected to be in logical order, any embedded directional introducers may exist in a code element string and are treated as any other code element by the multi-locale functions. It is only presentation-sensitive functions that transform between logical order text and visual order text that need to be aware of the embedded introducers. Such layout transformation functions are needed but are beyond the scope of this document.

In the future, multi-locale functions may need to be aware of the presentation order when performing their operation. For example the scan for next word function may need to be aware of the directionality of the text when doing its scanning; that is, the text may be in either visual order or logical order. While such functions may need to account for both a locale object and the layout directionality of the text; the multi-locale functions currently defined only operate on logical order text.

Language Context

Future standard encodings may include the ability to include introducers that specify the language of the following text. Handling of such encodings ultimately requires that the language context in the form of a locale object identifier be stored with each specific data segment Already, there are many applications (text editors) that associate a proprietary locale object with each language-sensitive data segment within their structured files. By providing support for multi-locales, the multi-locale functions allow applications to associate a locale object with a specific language segment, thus enabling multi-lingual applications. Applications are still responsible for segmenting any incoming text into whatever structured form they desire, and for associating a locale object with the specific segment.

If, in the future, a standard encoding and a standard opaque data type are defined, the multi-locale functions will still describe the proper set of capability needed to operate on that data type.

4.3 Data Types and Objects

The header **<mlocale.h>** declares all new data types, macros and multi-locale functions. The types declared are:

AttrObject
This is an opaque object type other than an array type that can hold values that represent the locale-specific information necessary for all locale categories.

LocaleSpec
This is an opaque object type other than an array type that can hold values that represent a locale specification. The content of a **LocaleSpec** is opaque but may be thought of as consisting of a **LocaleNetToken** or **LocaleNetString**. The **LocaleSpec** is the principle object used by programs within a host to announce a locale.

LocaleNetToken
This is an integer type that can hold any value corresponding to members of the Locale Registry as defined in Appendix A.

LocaleNetString
This is a **char*** type that can hold any values corresponding to members of the Locale Registry as defined in Appendix A.

Other data types objects defined by ISO, ANSI and the MSE standard are:

mbstate_t
This is an opaque object type other than an array type that can hold the locale-dependent stateful information necessary to convert, parse and tokenise code element strings. For example, this is used when converting between a coded character string and a wide-character string. The **mbstate_t** type is defined in the header **<wchar.h>**, as specified in the MSE standard.

wctrans_t
This is a scalar type that can hold values that represent the locale-specific transliteration mappings. The **wctrans_t** type is defined in the header **<wctype.h>**, as specified in the MSE standard.

wctype_t
This is a scalar type that can hold values that represent the locale-specific character classifications. The **wctype_t** type is defined in the header **<wctype.h>**, as specified in the MSE standard.

wchar_t
This is an integral type whose range of values can represent distinct wide-character codes for all members of the largest character set specified among the locales supported by the compilation environment: the null character has the code value zero, and each member of the Portable Character Set has a code value equal to its value when used as the lone character in an integer character constant.

With multiple locales, the value of a wide character is locale specific; it depends on the locale object (**AttrObject**) use to create the wide character. Wide character FOO created in locale A may be different from wide character FOO created in locale B. Any operation on a wide character is expected to use the same locale object used to create the wide character, otherwise the result is implementation dependent.

Refer to the MSE standard for the definitions of these data types.

4.3.1 Program Flow Model

The original global model of program flow is as follows:

1. Query locale.
2. Set locale to BAR if not already set.
3. Do operation FOO.
4. Restore locale.

This is replaced in the new model by:

1. Do operation FOO using locale BAR.

The fundamental programming paradigm is therefore altered to define localisation on a per-call rather than a per-process basis. In fact the model is generalised further by the *m_*( )* functions through the inclusion of an even more generic object. A locale object provides a holder for all manner of non-global data, of which the localised data is merely one example.

Note: All data types defined in this document are implementation defined, meaning that application developers should make no assumptions about their size, contents or capabilities.

4.3.2 Locale Object — AttrObject

The principal locale object is of type **AttrObject**, which is an opaque data structure that may contain several attributes.

A locale object provides an opaque data type that can be used to identify specific localisation requirements on a local (per-call) basis. This and the associated set of APIs that accept locale objects as input arguments satisfy the fundamental requirement for a set of APIs that support the operation of multi-lingual, multi-threaded applications in a distributed environment.

However, applications have other requirements for localised operation, such as security controls, which are outside the scope of this specification, but which (when addressed in the future) should not result in yet another set of APIs. Thus a locale object may be insufficient on its own to satisfy the system-wide requirements of multi-threaded applications. Therefore the data type **AttrObject** is proposed to be a generic object which is (or can be) a container of many opaque objects. A locale is just one example of the type of object that can be attached to an **AttrObject**; indeed, it is the only such object defined at present. Other objects may be defined, and associated with an **AttrObject**, as other working groups address the threads issue in more detail.

Throughout this document, several functions are defined to accept a locale object as an argument. In some cases the need for a locale object may not be obvious, yet it is included for consistency.

4.3.3 Text Context Object — mbstate_t

The MSE standard introduces the concept of an **mbstate_t** data type, called here a *text context object*. This is used with interfaces that need to retain state or other context-sensitive information between calls (for example, the stream I/O functions). Within a stateful text stream, context for characters are determined from the previous introducer, or possibly introducers (see Section 4.2.3 on page 23).

The MSE standard defines an **mbstate_t** object for the purpose of handling stateful encodings within streams. Specifically, it claims:

> ... each wide-oriented stream has an associated **mbstate_t** object that stores the current parse state of the stream ...

This specification extends the use of the **mbstate_t** object to enable it to be used with multi-locale functions to perform stream operations independent of the global locale and allows stream functions defined in the MSE standard to be associated with a locale object. If the **mbstate_t** object is created with the *m_creatembstate*( ) function, then the MSE functions taking an **mbstate_t** object use the locale associated with that **mbstate_t** instead of the global locale. When the *m_fattr*( ) function is called to associate a stream with an attribute object, that attribute object is used by I/O functions that take a **FILE** object.

Beyond conversion between coded characters and wide-character encodings, a text context object is viewed as capable of handling other contextual processing, for example the *m_wcstok*( ) function used to tokenise a string.

4.3.4 Classification Object — wctype_t

The classification object defines a set of locale-specific classifications at the time that it is created. The *m_wctype*( ) function allows a classification object to be instantiated using a locale object.

4.3.5 Transliteration Object — wctrans_t

The transliteration object defines a set of locale-specific transliterations at the time that it is created. The *m_wctrans*( ) function allows a transliteration object to be instantiated using a locale object.

4.3.6 Concurrency (Thread Safeness)

The multi-locale functions are designed to function properly in the presence of concurrent tasks sharing memory within an application, where more than one task is performing multi-locale operations at the same time on the same object; for example, **AttrObject**. The definition of *m_setlocale*( ) is to modify the locale object within the **AttrObject** and so it affects all threads sharing the same **AttrObject**.

The expected model is :

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

    ... do localised work using locale, possibly passing ‘locale’
        to helper threads

m_destroyattrobj(locale);
```

4.4 Distributed Locale Functions

The multi-locale functions provide the ability to manage the localised behaviour of functions across a network. Specifically, this document provides application developers with the following:

- a network locale specification that may be communicated over a network as either a token or string
- a locale specification that encapsulates the representation of a network locale specification on a host system
- functions to map between a locale specification and a host locale name
- a registry of locales that are expected to be found in a distributed environment.

Distributed Locale Functions
m_createlocspec( )
m_locspec_to_host( )
m_locspec_from_host( )
m_destroylocspec( )
m_locspec_to_nettoken( )
m_locspec_to_netstring( )
m_locspec_from_nettoken( )
m_locspec_from_netstring( )

Application developers are expected to use the locale specification (*m_locspec_**( )) communicating locales to underlying communication layers. The *m_locspec_to_host*( ) function retrieves a host locale name (representing all categories, LC_ALL) from a locale specification. The host locale name may be passed to either the *m_setlocale*( ) or *setlocale*( ) functions. The *m_locspec_from_host*( ) function is used to get a locale specification from a host locale name.

The local type **LocaleSpec** is an opaque data type that cannot be communicated over a network. The *m_locspec_to_nettoken*( ) and *m_locspec_to_netstring*( ) functions are used to convert a locale specification to a format that may be communicated over a network. The *m_locspec_from_nettoken*( ) and *m_locspec_from_netstring*( ) functions are used to convert from a network format (token or string) into a locale specification.

The *m_locspec_to_nettoken*( ) and *m_locspec_to_netstring*( ) functions are expected to be managed by underlying communication layers; for example, an RPC layer. The *m_locspec_from_nettoken*( ) and *m_locspec_from_netstring*( ) functions are used by any component needing to announce a host locale. Refer to *m_createlocspec*( ) on page 45 for examples of how to use these functions. Refer to Section 3.3 on page 14 for an overview of locale usage for the host and network.

The intended users of *m_locspec_**( ) functions are as follows:

Used by Applications	
AttrObject ↔ host_string ↔ **LocaleSpec** ←┤	→ **LocaleNetToken** → **LocaleNetString**
	Used by Communication Services

4.5 Locale Management Functions

A set of multi-locale functions are proposed that provide similar capabilities to those provided in the existing ISO C standard, the ISO POSIX-1 standard and the **XSH, Issue 4** specification. The naming convention for multi-locale function is to prefix them with *m_*. In cases where new capability is introduced, both a multi-locale and global-locale version is introduced.

There are sundry management functions associated with the creation and initialisation of objects that contain locale information within a particular host system. The locale management functions are as follows:

Global-locale	Multi-locale
setlocale( )	*m_createattrobj*( ) *m_destroyattrobj*( ) *m_setlocale*( )
mbsinit( )†	*m_creatembstate*( ) *m_destroymbstate*( ) *mbsinit*( )†
	m_fattr( )

Note: The function marked with a dagger (†) is defined by the MSE standard but is not included in the **XSH, Issue 4** specification.

The MSE standard does not define a method to initialise the object, whereas the functions defined in this document do, because of the non-global nature of locale objects. The MSE standard function *mbsinit*( ) allows you to query the initial state of an **mbstate_t** object. An **mbstate_t** object in a multi-locale environment may be used as follows:

```
ParseForLocalisedComment(AttrObject mylocale, char *str)
{
    mbstate_t mystate = m_creatembstate(my_locale);
    char *p = m_strtok(str, "\t", &mystate);
    p = m_strtok(str, "#", &mystate);
    m_destroymbstate(mystate);
    return(p);
}
```

The *m_creatembstate*( ) function is defined to allow a locale object to be associated within an **mbstate_t** object and must be freed by calling the *m_destroymbstate*( ) function.

4.6 Locale Information Functions

These functions provide access to locale-specific data associated with the LC_TIME, LC_MONETARY, LC_NUMERIC and LC_MESSAGES categories of a locale object. The locale information functions are as follows:

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
localeconv( )		*m_localeconv*( )	
nl_langinfo( )		*m_nl_langinfo*( )	
MB_CUR_MAX		*m_mb_cur_max*( )	
strerror( )		*m_strerror*( )	

4.7 Composite Character Sequence (CCS) Functions

These functions manipulate composite sequences. A *composite sequence* consists of a non-combining (base) character followed by one or more combining characters (floating diacritics). The non-combining character and the combining characters of the composite sequence are represented by corresponding code elements. The functions below are defined to work on strings that may consist of a single code element (non-combining character) or a composite sequence.

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
			m_wcscnt( )
			m_wcsnext( )
			m_wcsquery( )
	wcwidth( )		
	wcswidth( )‡		*m_wcswidth*( )‡

‡ This function may be used to determine the width of a single wide character.

4.8 Classification Functions

These functions provide for the classification of code elements. This includes a function similar to the *wctype*( ) function in the **XSH, Issue 4** specification, and a function similar to *m_iswctype*( ) for locating code elements of a particular type. There is also a more general scanning function (*m_wcsscanfor*( )) for locating code elements that match one or a set of classification criteria.

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
wctype( )	*wctype*( )	*m_wctype*( )	*m_wctype*( )
	iswctype( )	*m_isctype*( )	*m_iswctype*( )
*is**( )	*isw**( )		*m_iswctype*( )‡

‡ This function may be used in place of the *isw**( ) functions.

4.9 Transliteration Functions

The function *m_wctrans*( ) provides conversion operations similar to the *towupper*( ) function and *towlower*( ) functions in the **XSH, Issue 4** specification. The *m_towcstrans*( ), *towcstrans*( ) function are introduced to allow for transliterations of encodings with composite sequences that may alter the size of the output buffer size from the input buffer size.

Global locale		Multi-locale	
char	**wchar_t**	**char**	**wchar_t**
wctrans( )†	*wctrans*( )†	*m_wctrans*( )	*m_wctrans*( )
	towctrans( )†		
		m_tombstrans( )	*m_towcstrans*( )
toupper( )	*towupper*( )		
tolower( )	*towlower*( )		

Note: The functions marked with a dagger (†) are defined by the MSE standard but are not included in the **XSH, Issue 4** specification.

4.10 String Searching Functions

These functions make it possible to search for code elements within a code element string. They are similar to the standard string functions in the **XSH, Issue 4** specification.

Global locale		Multi-locale	
char	**wchar_t**	**char**	**wchar_t**
strpbrk( )	*wcspbrk*( )	*m_strpbrk*( )	*m_wcspbrk*( )
strspn( )	*wcsspn*( )	*m_strspn*( )	*m_wcsspn*( )
strcspn( )	*wcscspn*( )	*m_strcspn*( )	*m_wcscspn*( )
strstr( )	*wcswcs*( )	*m_strstr*( )	*m_wcswcs*( )

4.11 String Comparison Functions

These functions provide for the comparison of code elements within a code element string. They are similar to the standard string functions in the **XSH, Issue 4** specification.

Global locale		Multi-locale	
char	**wchar_t**	**char**	**wchar_t**
strcoll( )	*wcscoll*( )	*m_strcoll*( )	*m_wcscoll*( )
strxfrm( )	*wcsxfrm*( )	*m_strxfrm*( )	*m_wcsxfrm*( )

4.12 Date, Monetary and Time Formatting Functions

The date and time functions operate on text objects. These directly parallel functions in the **XSH, Issue 4** specification.

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
strftime( )	*wcsftime*( )	*m_strftime*( )	*m_wcsftime*( )
strptime( )	*wcsptime*( )	*m_strptime*( )	*m_wcsptime*( )
strfmon( )		*m_strfmon*( )	*m_wcsfmon*( )

4.13 Number Conversion Functions

The number conversion functions are as follows:

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
strtod( )	*wcstod*( )	*m_strtod*( )	*m_wcstod*( )
strtol( )	*wcstol*( )	*m_strtol*( )	*m_wcstol*( )
strtoul( )	*wcstoul*( )	*m_strtoul*( )	*m_wcstoul*( )

4.14 Text Scanning and Parsing Functions

These functions enable applications to scan and parse localised strings (of code elements) into tokens. If necessary, a text context object is used to preserve any needed context across multiple calls to these functions.

The following text parsing functions are currently defined:

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
strtok( )	*wcstok*( )	*m_strtok*( )	*m_wcstok*( )
		m_strscanfor( )‡	*m_wcsscanfor*( )‡

‡ These functions do not have global versions. As with all *m_*( ) functions, they can be used with the *attrobj* argument NULL.

4.15 Text Formatted I/O Functions

This specification extends the *printf* and *scanf* utilities to enable them to use an explicit locale.

The MSE standard introduced the concept of wide and narrow I/O streams. A wide stream has an implicit **mbstate_t** object associated with the stream. This object is used when parsing strings. This specification introduces the *m_fattr*( ) function which enables an attribute object to be explicitly associated with both wide and narrow streams. If the *m_setlocale*( ) function has been successfully called to associate a locale with the attribute object and that attribute object is associated with the stream with the *m_fattr*( ) function, then this explicit locale is used by I/O functions operating on the stream. Otherwise, the behavior is as described by the existing ISO C standard or MSE standard for narrow and wide streams respectively.

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 multi-byte encodings valid for use internal to the program).
- A file need not begin or end in the initial state.[2] Moreover, the encodings used for encoding of text in external 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 choice of such encodings are implementation defined.

The following functions may be used in multi-locale environments when the stream's **mbstate_t** object is associated with a specific locale object.

Global locale		**Multi-locale**	
char	**wchar_t**	**char**	**wchar_t**
printf( )	*wprintf*( )†	*printf*( )‡	*wprintf*( )‡
scanf( )	*wscanf*( )†	*scanf*( )‡	*wscanf*( )‡
fprintf( )	*fwprintf*( )†	*fprintf*( )‡	*fwprintf*( )‡
sprintf( )	*swprintf*( )†	*m_sprintf*( )	*m_swprintf*( )
fscanf( )	*fwscanf*( )†	*fscanf*( )‡	*fwscanf*( )‡
sscanf( )	*swscanf*( )†	*m_sscanf*( )	*m_swscanf*( )
fgetc( )	*fgetwc*( )	*fgetc*( )‡	*fgetwc*( )‡
fgets( )	*fgetws*( )	*fgets*( )‡	*fgetws*( )‡
fputc( )	*fputwc*( )	*fputc*( )‡	*fputwc*( )‡
fputs( )	*fputws*( )	*fputs*( )‡	*fputws*( )‡
ungetc( )	*ungetwc*( )	*ungetc*( )‡	*ungetwc*( )‡

Notes:

1. The functions marked with a dagger (†) are defined by the MSE standard but are not included in the **XSH, Issue 4** specification. It is assumed that they will be included in a future issue, when XSH is aligned with the MSE standard.

2. Setting the file position indicator to end-of-file, as with *fseek*(file, 0, SEEK_END), has undefined behaviour for a binary stream (because of possible trailing null characters) or for any stream with state-dependent encoding that does not assuredly end in the initial shift state. For streams with state dependent encodings, if the file does not begin or end in the initial shift state, it is the responsibility of the application to associated an **mbstate_t** whose state corresponds to the state of the position in the file.

2. The functions marked with a double dagger (‡) indicate that the functions may be used in a multi-locale fashion if the **mbstate_t** has been initialised using the *m_fattr*( ) function.

For the multi-locale functions, all locale-specific processing (conversion, parsing, numeric formatting) is associated with the locale object contained in the stream's **mbstate_t** object.

4.16 Extended Wide-character Conversion Functions

These functions provide sufficient capability for converting a code element string from one class of encoding to another class of encoding, for example, from coded character string to wide-character strings. These functions are aligned with the corresponding functions described in the MSE standard.

These functions differ from the corresponding **XSH, Issue 4** specification functions (*mblen*( ), *mbtowc*( ), *wctomb*( )) in that they have an extra argument. This argument is *ps*, of type pointer to **mbstate_t**, which points to an object that can completely describe the current conversion state of the code element sequence.

Global locale	Multi-locale
mblen( )	*mbrlen*( )†
mbtowc( )	*mbrtowc*( )†
mbstowcs( )	*mbsrtowcs*( )†
wctomb( )	*wcrtomb*( )†
wcstombs( )	*wcsrtombs*( )†

Note: The functions marked with a dagger (†) are not currently published in the **XSH, Issue 4** specification. It is assumed that they will be included in a future issue, when XSH is aligned with the MSE standard.

If the **mbstate_t** object passed to one of the multi-locale functions was created with *m_creatembstate*( ) using an attribute object which has a valid locale associated with it by *m_setlocale*( ), then that explicit locale is used. Otherwise, if the **mbstate_t** object was not created by *m_creatembstate*( ), or if the attribute object does not have an associated locale, then the global locale is used.

Chapter 5

Header File

This chapter defines the header file **<mlocale.h>**.

NAME

mlocale.h — multi-locale macros

SYNOPSIS

```
#include <mlocale.h>
```

DESCRIPTION

The **<mlocale.h>** header provides a definition for objects:

```
typedef struct _attr_object_t *AttrObject;
typedef struct _sil_spec_t LocaleSpec;
typedef unsigned int LocaleNetToken;
typedef char* LocaleNetString;
```

The structures **_attr_object_t** and **_sil_spec_t** are opaque data structures that are implementation-dependent.

The **<mlocale.h>** header defines the following values for the text parsing function *m_wcsscanfor*( ).

```
#define NoCondition 0L
#define Alphabetic         (1L<<1)  /* same as isalpha */
#define WhiteSpace         (1L<<2)  /* same as isspace */
#define Control            (1L<<3)  /* same as iscntrl */
#define Digit              (1L<<4)  /* same as isdigit */
#define Graphic            (1L<<5)  /* same as isgraph */
#define Lowercase          (1L<<6)  /* same as islower */
#define Uppercase          (1L<<7)  /* same as isupper */
#define Printing           (1L<<8)  /* same as isprint */
#define Punctuation        (1L<<9)  /* same as ispunct */
#define HexDigit           (1L<<10) /* same as isxdigit */
#define LineBreakCharacter (1L<<11) /* a wide-character code that is
                                       defined to cause a line
                                       discontinuity */
#define LineBreakHyphen    (1L<<12) /* the next wide-character code
                                       after which a line discontinuity
                                       may occur due to hyphenation */
#define LineBreakScript    (1L<<13) /* the next wide-character code
                                       after which a line discontinuity
#define WordBoundary       (1L<<14) /* the next wide-character code
                                       after which the current language
#define SentenceBoundary   (1L<<15) /* the next wide-character code
                                       after which the current sentence */
#define ParagraphBoundary  (1L<<16) /* the next wide-character code
                                       after which the current paragraph */
#define CharsetBoundary    (1L<<17) /* the next wide-character code
                                       after which the current charset */
#define ScriptBoundary     (1L<<18) /* the next wide-character code
                                       after which the current script */
#define CompositeBoundary  (1L<<19) /* the next wide-character code
                                       after which the current composite
```

The **<mlocale.h>** header defines the following value for the *m_locspec_to_nettoken*( ) function:

```
#define NoLocaleNetToken -1
```

The **<mlocale.h>** header defines the following types through **typedef**:

ScanCondition
ScanDirection
Boolean

The **<mlocale.h>** header declares the following functions:

```
AttrObject m_createattrobj(void);
LocaleSpec *m_createlocspec(void);
mbstate_t m_creatembstate(AttrObject attrobj);
int m_destroyattrobj(AttrObject attrobj);
int m_destroylocspec(LocaleSpec *locspec);
int m_destroymbstate(mbstate_t ps);
int m_sprintf(mbstate_t *ps, char *s, const char *format, ...);
int m_sscanf(mbstate_t *ps, const char *s, const char *format, ... );
AttrObject m_fattr(FILE *stream, const AttrObject attrobj);
int m_isctype(const AttrObject attrobj, char* s, wctype_t desc);
int m_iswctype(const AttrObject attrobj, wint_t wc, wctype_t desc);
struct lconv *m_localeconv(const AttrObject attrobj);
int m_locspec_from_host(LocaleSpec locspec, const char* s);
int m_locspec_from_netstring(LocaleSpec locspec,
                   const LocaleNetString s);
int m_locspec_from_nettoken(LocaleSpec locspec,
                   const LocaleNetToken token);
LocaleNetString m_locspec_to_netstring(const LocaleSpec locspec);
char* m_locspec_to_host(const LocaleSpec locspec);
LocaleNetToken m_locspec_to_nettoken(const LocaleSpec locspec);
int m_mb_cur_max(const AttrObject *attrobj);
char *m_nl_langinfo(const AttrObject attrobj, nl_item item, char* buf,
                   size_t bufsize);
char *m_setlocale(AttrObject *attrobj, const int category,
                   const char *locale);
int m_strcoll(const AttrObject attrobj, const char *s1,
                   const char *s2);
size_t m_strcspn(const AttrObject attrobj, const char **str1,
                   const char **str2);
char *m_strerror(const AttrObject attrobj, int errnum);
ssize_t m_strfmon(const AttrObject attrobj, char *s, size_t maxsize,
                   const char *format, ...);
size_t m_strftime(const AttrObject attrobj, char *s, size_t maxsize,
                   const char *format, const struct tm *timptr);
char *m_strpbrk(const AttrObject attrobj, const char* *str1,
                   const char *str2);
char *m_strptime(const AttrObject attrobj, const char *buf,
                   const char *format, struct tm *tm);
size_t m_strscanfor(const AttrObject attrobj, const char* s,
                   size_t num_bytes, size_t position,
                   ScanDirection direction, ScanCondition condition,
                   Boolean inverse);
size_t m_strspn(const AttrObject attrobj, const char **str1,
                   const char **str2);
char *m_strstr(const AttrObject attrobj, const char *str1,
                   const char *str2);
```

```
double m_strtod(const AttrObject attrobj, const char *str,
                   char **endptr);
char *m_strtok(char *s1, const char *s2, mbstate_t *ps);
long int m_strtol(const AttrObject attrobj, const char *str,
                   char **endptr, int base);
unsigned long int m_strtoul(const AttrObject attrobj, const char *str,
                   char **endptr, int base);
size_t m_strxfrm(const AttrObject attrobj, char *s1,
                   const char *s2, size_t n);
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);
size_t m_wcscnt(const AttrObject attrobj, const wchar_t *ptr);
int m_wcscoll(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
size_t m_wcscspn(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
size_t m_wcsfmon(const AttrObject attrobj, wchar_t *ws,
                   size_t maxsize, const char *format, ...);
size_t m_wcsftime(const AttrObject attrobj, wchar_t *wcs,
                   size_t maxsize, wchar_t *format,
                   const struct tm *timptr);
size_t m_wcsnext(const AttrObject attrobj, const wchar_t *ptr);
wchar_t *m_wcspbrk(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
wchar_t *m_wcsptime(const AttrObject attrobj, const wchar_t *ws,
                   const char *format, struct tm *timptr);
size_t m_wcsquery(const AttrObject attrobj, const wchar_t *ptr);
size_t m_wcsscanfor(const AttrObject attrobj, const wchar_t* ws,
                   size_t num_chars, size_t position,
                   ScanDirection direction, ScanCondition condition,
                   Boolean inverse);
size_t m_wcsspn(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
double m_wcstod(const AttrObject attrobj, const wchar_t *src,
                   wchar_t **end_ptr);
wchar_t *m_wcstok(wchar_t *ws1, const wchar_t *ws2, mbstate_t* ps)
long m_wcstol(const AttrObject attrobj, const wchar_t *src,
                   wchar_t **end_ptr, int base)
unsigned long m_wcstoul(const AttrObject attrobj, const wchar_t *src,
                   wchar_t *endptr, int base);
wchar_t *m_wcswcs(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
size_t m_wcswidth(const AttrObject attrobj, const wchar_t *ptr,
                   size_t n);
size_t m_wcsxfrm(const AttrObject attrobj, wchar_t *ptr1,
                   const wchar_t *ptr2, size_t n);
wctrans_t m_wctrans(const AttrObject attrobj, const char *property);
wctype_t m_wctype(const AttrObject attrobj, const char *property);
```

SEE ALSO

m_localeconv( ), *m_setlocale*( ), the chapter on **Environment Variables** in the **XBD, Issue 4** specification.

Chapter 6

Reference Manual Pages

This chapter contains reference manual pages for the *m_*( )* functions and other wide-character or multi-byte functions required by *m_*( )* functions.

NAME

m_createattrobj — create locale attribute object

SYNOPSIS

```
#include <mlocale.h>

AttrObject m_createattrobj(void);
```

DESCRIPTION

The *m_createattrobj*( ) function returns an object of type **AttrObject** that contains the default attributes to be used in a multi-locale program. The returned object is guaranteed to be initialised with a locale attribute that identifies the appropriate pieces of an implementation-dependent locale. A locale attribute consist of the following locale categories: LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC and LC_TIME.

The categories of a locale attribute may be changed and queried by using the *m_setlocale*( ) function. Other attributes may be defined by implementations but are beyond the scope of this specification.

RETURN VALUE

Upon successful completion, the *m_createattrobj*( ) function returns a locale attribute object for use in subsequent calls to multi-locale, *m_**( ), functions. Otherwise *m_createattrobj*( ) returns (**AttrObject**)0 and sets *errno* to indicate the error.

ERRORS

The *m_createattrobj*( ) function may fail if:

[ENOMEM]
Insufficient storage space is available.

SEE ALSO

m_destroyattrobj( ), *m_setlocale*( ), **<mlocale.h>**.

CHANGE HISTORY

Derived from Version 1 of this document.

NAME

m_createlocspec — allocates and returns a locale specification

SYNOPSIS

```
#include <mlocale.h>

LocaleSpec  *m_createlocspec(void);
```

DESCRIPTION

The *m_createlocspec*( ) function returns an object of type **LocaleSpec** that contains the locale specification in a system-independent form; this is used to distribute locales over a network. The returned **LocaleSpec** object is defined to be empty and not associated with any locale. The caller should call the *m_locspec_from_netstring*( ), *m_locspec_from_nettoken*( ) or *m_locspec_from_host*( ) functions to initialise the **LocaleSpec** object with a network locale specification. The content of a **LocaleSpec** is implementation dependent but must be associated with a network locale specification.

The *m_fattr*( ) function queries the locale of the **AttrObj** passed in, using the returned value to create a clone of that **AttrObj**. The passed-in object is not bound to or modified by any stream operation.

RETURN VALUE

Upon successful completion, the *m_createlocspec*( ) function returns an object of type **LocaleSpec** for use in subsequent calls to distributed locale, *m_locspec**( ), functions. Otherwise the *m_createlocspec*( ) function returns (**LocaleSpec**)0 and sets *errno* to indicate the error.

ERRORS

The *m_createlocspec*( ) function may fail if:

[ENOMEM] Insufficient storage space is available.

APPLICATION USAGE

The following is an example of a client program using a system-independent locale specification:

```
/*** CLIENT is application-dependent handle, ***/
/*** for example, clnt_create(...)           ***/

void foo(AttrObject client_locale,
           CLIENT*         client_handle);
{
   char*       host_string = m_setlocale(client_locale, LC_ALL, NULL);
   LocaleSpec* locspec = m_createlocspec();

   if ( m_locspec_from_host(locspec, host_string) == -1 ) {
       /*
        * Client’s locale is NOT known to the network,
        * ... report error ...
        */
       return ;
   }
   rpc_foo_operation(locspec, client_handle);
   m_destroylocspec(locspec);
}
```

On the client side, the *rpc_foo_operation*( ) function is responsible for marshalling the **LocaleSpec** into a protocol-defined network locale specification (**LocaleNetToken** or **LocaleNetString**) that is passed to the server process. The *m_locspec_to_nettoken*( ) or *m_locspec_to_netstring*( ) functions are used to obtain a well known data type depending on the protocol definition.

On the server side, the protocol-defined network locale specification is marshalled into a **LocaleSpec** using the *m_locspec_from_nettoken*( ) or *m_locspec_from_netstring*( ) functions.

The following is an example of a server program using a system-independent locale specification:

```
rpc_foo_operation(LocaleSpec locspec, struct svc_req *req)
{
   char           *host_string;
   AttrObject      host_locale;

   if ( ! (host_string = m_locspec_to_host(locspec)) ) {
       /*
        * Network locale is NOT known to the host server
        * ... report error ...
        */
   }

   host_locale = m_createattrobj();
   if (m_setlocale(host_locale, LC_ALL,
       host_string) == (AttrObject)NULL ) {
       /*
        * Locale is NOT available to the server
        * ... report error ...
        */
   };
   free(host_string);

   foo(host_locale);  /* do real work */
   m_destroyattrobj(host_locale);
}
```

For clarity, the above examples are simple. Real distributed applications often need to specify which locale category a particular operation must use. Consider that a **LocaleSpec** is made up of a well defined locale where all the categories are registered as a whole for network distribution. Thus a locale object made up of mixed categories needs to be broken down into separate categories.

Application developers that need to use different locale categories within a locale object must design their RPC calls to pass multiple locale specifications per category. For example, an RPC may be defined to do the operation in locale FOO but to provide messages in locale BAR. A server may construct a single **AttrObject** made up of multiple **LocaleSpec** types. The following is an example of a server that takes two **LocaleSpec** types to perform operation bar.

```
rpc_bar_operation(LocaleSpec locspec, LocaleSpec msgspec,
                  struct svc_req *req)
{
   char           *host_string = m_locspec_to_host(locspec);
   char           *msg_string;
   AttrObject      host_locale;
   AttrObject      msg_locale;

   /*
```

```
         * Establish Message Locale first
         */
        msg_string = m_locspec_to_host(msgspec);
        msg_locale = m_createattrobj();
        m_setlocale( msg_locale, LC_ALL, msg_string);
           /*
            * If m_setlocale fails, message locale is NOT available
            * at the server, then fall back to implementation default...
            */
        free(msg_string);

        if ( ! host_string ) {
            /*
             * Locale is NOT known to the server
             * ... report error ...
             */
            errstring = m_strerror( msg_locale, errno );
            rpc_bar_error( msgspec, errstring, req);
            m_destroyattrobj(msg_locale);
            return;
        }

        host_locale = m_createattrobj();
        if (m_setlocale(host_locale, LC_ALL,
            host_string) == (AttrObject)NULL ) {
            /*
             * Locale is NOT available to the server
             * ... report error ...
             */
            errstring = m_strerror( msg_locale, errno );
            rpc_bar_error( msgspec, errstring, req);
            m_destroyattrobj(msg_locale);
            m_destroyattrobj(host_locale);
            return;
        };
        free(host_string);

        /*
         * Set message category of host_locale to message locale
         */
        m_setlocale( host_locale,
                     LC_MESSAGES,
                     m_setlocale( msg_locale, LC_MESSAGES, NULL));

        foo(host_locale);  /* do real work */
        m_destroyattrobj(msg_locale);
        m_destroyattrobj(host_locale);
    }
```

SEE ALSO

m_locspec_to_host( ), *m_locspec_from_host*( ), *m_locspec_to_nettoken*( ), *m_locspec_from_nettoken*( ), *m_locspec_to_netstring*( ), *m_locspec_from_netstring*( ), *m_destroylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_creatembstate — create text context object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

mbstate_t m_creatembstate(const AttrObject attrobj);
```

DESCRIPTION

The *m_creatembstate*( ) function returns an object of type **mbstate_t** which may contain state information associated with the locale attribute identified by *attrobj* for use with contextual text. For state-dependent encodings, the text context object is placed into a codeset-dependent initial state, ready for immediate use with functions requiring contextual state handling.

The returned text context object maintains an import conversion state when used with the following functions defined in the MSE standard: *mbsrtowcs*( ), *mbrtowc*( ), *mbrlen*( ), *fgetwc*( ), *fgetws*( ), *fwscanf*( ) and *m_swscanf*( ). These functions use the locale attribute associated with attrobject.

The returned text context object may maintain an export conversion state when using the following functions: *wcsrtombs*( ), *wcrtomb*( ), *fputws*( ), *fwprintf*( ), *m_sprintf*( ) and *m_swprintf*( ). These functions use the locale attribute associated with attrobject.

The returned text context object maintains a tokenising context when using the *m_wcstok*( ) function.

Other contextual states may be defined by implementations but are beyond the scope of this specification.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_creatembstate*( ) function returns a text context object that is in the initial state for all the possible conversions and tokenisation. Otherwise, *m_creatembstate*( ) returns (**mbstate_t**)0 and sets errno to indicate the error.

ERRORS

The *m_creatembstate*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[ENOMEM]
Insufficient storage space is available.

APPLICATION USAGE

The MSE standard declares in clause 4.6.5 that if the **mbstate_t** object has been altered by any of the functions described in this subclause of the MSE standard, and is then used with a different multi-byte character sequence, or *in the other conversion direction*, or with a *different LC_CTYPE category setting from the earlier function calls*, the behaviour is undefined. Yet, when an **mbstate_t** is created using the *m_creatembstate_t*( ) function, the behaviour is well defined using the locale attribute identified by *attrobj*.

The MSE standard may imply that the **mbstate_t** object can only maintain a single state (for import or export) at any given time. For the **XSH, Issue 4** specification, the behaviour is defined that the returned object of type **mbstate_t** will be capable of handling multiple states (for import, export or tokenising) within a single **mbstate_t** object.

SEE ALSO

m_createattrobj( ), *m_setlocale*( ), *m_destroymbstate*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_destroyattrobj — destroy locale attribute object

SYNOPSIS

```
#include <mlocale.h>

int m_destroyattrobj(AttrObject *attrobj);
```

DESCRIPTION

The *m_destroyattrobj*( ) function deallocates the locale attribute object *attrobj* and all other associated resources allocated by the *m_createattrobj*( ), or *m_setlocale*( ) function.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

The *m_destroyattrobj*( ) function may fail if:

[EBADF]
The attribute object is invalid.

SEE ALSO

m_createattrobj( ), *m_setlocale*( ), **<mlocale.h>**.

CHANGE HISTORY

Derived from Version 1 of this document.

NAME

m_destroylocspec — allocates and returns a network locale specification

SYNOPSIS

```
#include <mlocale.h>

int m_destroylocspec(LocaleSpec *locspec);
```

DESCRIPTION

The *m_destroyattrobj*( ) function deallocates the **LocaleSpec** object and all other associated resources allocated by the *m_createlocspec*( ) function.

RETURN VALUE

Upon successful completion, the *m_destroylocspec*( ) function returns a value of zero. Otherwise a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

The *m_destroylocspec*( ) function may fail if:

[EBADF]
: The **LocaleSpec** object is invalid.

APPLICATION USAGE

Refer to *m_createlocspec*( ) for examples.

SEE ALSO

m_createlocspec( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_destroymbstate — destroy text context object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

int m_destroymbstate(mbstate_t *ps);
```

DESCRIPTION

The *m_destroymbstate*( ) function deallocates the text context object *ps* and all other associated resources allocated by the *m_creatembstate*( ) function.

RETURN VALUE

Upon successful completion, a value of zero is returned. Otherwise a value of –1 is returned and *errno* is set to indicate the error.

ERRORS

The *m_destroymbstate*( ) function may fail if:

[EBADF]
The **mbstate_t** object is invalid.

SEE ALSO

m_creatembstate( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_fattr — query or set locale of a stream using a locale object

SYNOPSIS

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

DESCRIPTION

The *m_fattr*( ) function will associate an explicit **mbstate_t** object whose locale is defined by *attrobj* with stream. If the **mbstate_t** is successfully created and associated with stream, then the **AttrObject** used to create the **mbstate_t** is returned.

The returned **AttrObject** is a copy of the locale object in the **mbstate_t** of stream, and the application should use the *m_destroymbstate*( ) function to free up the returned **AttrObject**.

If *attrobj* is defined as (**AttrObject**)NULL, the *m_fattr*( ) function will return an **AttrObject** of the **mbstate_t** associated with stream.

RETURN VALUE

The *m_fattr*( ) function returns the **AttrObject** of the **mbstate_t** associated with stream. If unsuccessful, (*AttrObject*)NULL is returned and *errno* is set.

ERRORS

The *m_fattr*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
If *attrobj* is (**AttrObject**)NULL and stream has not been initialised with an **mbstate_t**, or if *attrobj* is not (**AttrObject**)NULL and the stream has been previously initialised with an **mbstate_t**.

APPLICATION USAGE

Since it is not known whether the returned **AttrObject** was created with the *m_creatembstate*( ) function or declared initially as a zero-value **mbstate_t** object, the caller must always assume the worst and issue the *m_destroymbstate*( ) function.

SEE ALSO

m_creatembstate( ), *m_setlocale*( ), *m_destroymbstate*( ), **<mlocale.h>**.

NAME

m_isctype — test coded character for specified class using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wctype.h>

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

DESCRIPTION

The *m_isctype*( ) function determines whether the first coded character, as if by a call to *mbrtowc*( ), pointed to by *s* has the character class *desc*, returning true or false. The setting of the LC_CTYPE category in the *attrobj* shall be the same as during the call to *m_wctype*( ) or *wctype*( ) that returned the value *desc*. Otherwise the result is implementation dependent.

The *m_isctype*( ) function is defined to operate on coded characters corresponding to the valid character encodings in the locale defined in *attrobj*. If the coded character pointed to by the *s* argument is not in the domain of the locale defined by *attrobj*, the result is undefined.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_isctype*( ) function returns non-zero (true) if and only if the value of the coded character *s* has the property described by *desc*, otherwise returns zero (false).

ERRORS

The *m_isctype*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *iswctype*( ) function when called with the current locale set to the locale defined by *attrobj*.

The twelve strings — "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" — are reserved for the standard character classes. See *m_iswctype*( ) for the semantics of the reserved strings.

SEE ALSO

m_wctype( ), *m_iswctype*( ), *wctype*( ), *isalnum*( ), *isalpha*( ), *iscntrl*( ), *isdigit*( ), *isgraph*( ), *islower*( ), *isprint*( ), *ispunct*( ), *isspace*( ), *isupper*( ), *isxdigit*( ), **<mlocale.h>**, **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_iswctype — test wide character for specified class using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wctype.h>

int m_iswctype(const AttrObject attrobj, wint_t wc, wctype_t desc);
```

DESCRIPTION

The *m_iswctype*( ) function determines whether the wide-character code *wc* has the character class *desc*, returning true or false. The setting of the LC_CTYPE category in the *attrobj* shall be the same as during the call to *m_wctype*( ) or *wctype*( ) that returned the value *desc*. Otherwise the result is implementation dependent.

The *m_iswctype*( ) function is defined to operate on WEOF and wide-character codes corresponding to the valid character encodings in the locale defined in *attrobj*. If the *wc* argument is not in the domain of the locale defined by *attrobj*, the result is undefined.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_iswctype*( ) function returns non-zero (true) if and only if the value of the wide character *wc* has the property described by *desc*, otherwise returns zero (false).

ERRORS

The *m_iswctype*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *iswctype*( ) function when called with the current locale set to the locale defined by *attrobj*. The twelve strings — "alnum", "alpha", "blank", "cntrl", "digit", "graph", "lower", "print", "punct", "space", "upper" and "xdigit" — are reserved for the standard character classes. In the table below, the functions in the left column are equivalent to the functions in the right column for the same LC_CTYPE category setting.

```
iswalnum(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("alnum")))
iswalpha(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("alpha")))
iswcntrl(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("cntrl")))
iswdigit(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("digit")))
iswgraph(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("graph")))
iswlower(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("lower")))
iswprint(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("print")))
iswpunct(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("punct")))
iswspace(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("space")))
iswupper(wc)    m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("upper")))
iswxdigit(wc)   m_iswctype(attrobj, wc, m_wctype(attrobj, wctype("xdigit")))
```

Note: The call:

```
m_iswctype(attrobj, wc, m_wctype(attrobj, "blank"))
```

does not have an equivalent *isw**( ) function.

SEE ALSO

m_wctype( ), *wctype*( ), *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), **<mlocale.h>**. **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_localeconv — determine number formatting information using locale object

SYNOPSIS

```
#include <mlocale.h>

struct lconv *m_localeconv(const AttrObject attrobj);
```

DESCRIPTION

The *m_localeconv*( ) function sets the components of an object with the type **struct lconv** with the values appropriate for the formatting of numeric quantities (monetary and otherwise) according to the rules of the locale identified by *attrobj*.

The members of the structure with type **char*** are pointers to strings, any of which (except decimal point) can point to "", to indicate that the value is not available in the locale identified by *attrobj* or is zero length. The members with type **char** are non-negative numbers, any of which can be {CHAR_MAX} to indicate that the value is not available in the locale identified by *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_localeconv*( ) function returns a pointer to the filled-in object. The memory used for the filled-in object is allocated and managed by the argument *attrobj*. The structure pointed to by the return value must not be modified by the program. Its contents remain unchanged unless:

- The *attrobj* is modified by calling the *m_setlocale*( ) function with the categories LC_ALL, LC_MONETARY or LC_NUMERIC with the same *attrobj*.
- The *attrobj* is destroyed.

In either case, the content of the structure is undefined.

ERRORS

The *m_localeconv*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *localeconv*( ) function when called with the program’s locale set to the locale identified by *attrobj*. The **lconv** structure contains the same members as described for *localeconv*( ) in the **XSH, Issue 4** specification.

Members of the structure with type **char*** are encoded in the codeset of the locale identified by *attrobj*. If a wide-character form of the data is needed, it should be converted from file coded characters form to wide-character form using the *mbsrtowcs*( ) function.

This function returns a pointer to a string managed by *attrobj*. This requires that applications needing thread-specific data while sharing a single *attrobj* manage concurrent access to the *attrobj*.

SEE ALSO

m_createattrobj( ), *strchr*( ), *strcoll*( ), *strftime*( ), *strpbrk*( ), *strspn*( ), *strtok*( ), *strxfrm*( ), *strtod*( ), *localeconv*( ), **<langinfo.h>**, **<locale.h>**, **<mlocale.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_locspec_from_host — set locale specification using host locale string

SYNOPSIS

```
#include <mlocale.h>

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

DESCRIPTION

The *m_locspec_from_host*( ) function matches the host locale string to a network locale specification. If the host locale string is recognised as a network locale specification the *locspec* object is set with implementation-dependent information (for example, **LocaleNetToken**). When communicating the locale specification over a network use the *m_locspec_to_nettoken*( ) or *m_locspec_to_netstring*( ) functions to convert the **LocaleSpec** into a well known type.

RETURN VALUE

Upon successful completion, the *m_locspec_from_host*( ) function returns zero. Otherwise the *m_locspec_from_host*( ) function returns –1, sets *errno* to indicate the error, and *locspec* is unchanged. If the host locale string is not recognised as a network locale specification, an error is returned.

ERRORS

The *m_locspec_from_host*( ) function may fail if:

[EBADF]
  The **LocaleSpec** object is invalid.

[EINVAL]
  The host locale string *s* is not known as a network locale specification.

APPLICATION USAGE

If the host locale string is associated with a mixture of locale categories obtained from:

```
m_setlocale(attrobj, LC_ALL, NULL)
```

then all the categories specified by the host locale string, *s*, must be associated with the same network locale specification.

SEE ALSO

m_locspec_to_host( ), *m_createlocspec*( ), *m_destroylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_locspec_from_netstring — set locale specification using host locale string

SYNOPSIS

```
#include <mlocale.h>

int m_locspec_from_netstring(LocaleSpec *locspec,
                              const LocaleNetString s);
```

DESCRIPTION

The *m_locspec_from_netstring*( ) function converts the network locale specification defined by *s* into a system-independent form that is stored in the *locspec* object. The internal form of the the *locspec* object is implementation dependent.

The *m_locspec_from_netstring*( ) function is intended for low-level protocol communication services, for example, RPC run-time services, that need to convert the locale specification into a well known type. Most applications are expected to convert the locale specification into an **AttrObject** locale object for processing.

RETURN VALUE

Upon successful completion, the *m_locspec_from_netstring*( ) function returns zero. Otherwise the *m_locspec_from_netstring*( ) function returns −1, sets *errno* to indicate the error, and the *locspec* is unchanged.

ERRORS

The *m_locspec_from_netstring*( ) function may fail if:

[EBADF]
The **LocaleSpec** object is invalid.

[EINVAL]
The supplied network locale specification *s* is invalid.

SEE ALSO

m_locspec_from_nettoken( ), *m_locspec_to_netstring*( ), *m_locspec_to_nettoken*( ), *m_createlocspec*( ), *m_destroylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_locspec_from_nettoken — set locale specification using locale token

SYNOPSIS

```
#include <mlocale.h>

int m_locspec_from_nettoken(LocaleSpec *locspec,
                            const LocaleNetToken token);
```

DESCRIPTION

The *m_locspec_from_nettoken*( ) function converts the network locale specification defined by *token* into a system-independent form that is stored in the *locspec* object. The internal form of the *locspec* object is implementation dependent.

The *m_locspec_from_nettoken*( ) function is intended for low-level protocol communication services, for example, RPC run-time services, that need to convert the *locspec* into a well known type. Most applications are expected to convert the *locspec* into an **AttrObject** locale object for processing.

RETURN VALUE

Upon successful completion, the *m_locspec_from_nettoken*( ) function returns zero. Otherwise the *m_locspec_from_nettoken*( ) function returns −1, sets *errno* to indicate the error, and the *locspec* is unchanged.

ERRORS

The *m_locspec_from_nettoken*( ) function may fail if:

[EBADF]
: The **LocaleSpec** object is invalid.

[EINVAL]
: The supplied network locale specification *token* is invalid.

SEE ALSO

m_locspec_from_netstring( ), *m_locspec_to_netstring*( ), *m_locspec_to_nettoken*( ), *m_createlocspec*( ), *m_destroylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_locspec_to_netstring — get string network locale specification using locale specification

SYNOPSIS

```
#include <mlocale.h>

LocaleNetString m_locspec_to_netstring(const LocaleSpec locspec);
```

DESCRIPTION

The *m_locspec_to_netstring*( ) function returns the string network locale specification associated with the object *locspec* of type **LocaleSpec**. The returned string corresponds to a network locale specification associated with the **LocaleSpec** (as specified by category LC_ALL) that may be distributed over networks.

The returned string should be freed using the *free*( ) function.

The *m_locspec_to_netstring*( ) function is intended for low-level protocol communication services, for example, RPC run-time services, that need to convert the locale specification into a well known type. Most applications are expected to convert the locale specification into an **AttrObject** locale object for processing.

RETURN VALUE

Upon successful completion, the *m_locspec_to_netstring*( ) function returns a pointer to a string representing the network locale specification. Otherwise the *m_locspec_to_netstring*( ) function returns NULL and sets *errno* to indicate the error.

ERRORS

The *m_locspec_to_netstring*( ) function may fail if:

[EBADF]
The **LocaleSpec** object is invalid.

[EINVAL]
The **LocaleSpec** specified by *locspec* cannot be represented as a **LocaleNetString**. This can occur when the **LocaleSpec** contains a network locale specification in a token form but the corresponding string form cannot be found.

APPLICATION USAGE

Refer to the *m_createlocspec*( ) function for a description of the use of this function.

SEE ALSO

m_locspec_from_netstring( ), *m_createlocspec*( ), *m_destorylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_locspec_to_host — get host locale string using locale specification

SYNOPSIS

```
#include <mlocale.h>

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

DESCRIPTION

The *m_locspec_to_host*( ) function matches the network locale specification to a host locale string. If the network locale specification is recognised as a host locale, a fully qualified host locale string (as specified by category LC_ALL) is returned. The returned host locale string may be used locally with the *setlocale*( ) or *m_setlocale*( ) functions. The returned string should be freed using the *free*( ) function.

RETURN VALUE

Upon successful completion, the *m_locspec_to_host*( ) function returns a pointer to a string representing the host locale string. Otherwise the *m_locspec_to_host*( ) function returns NULL and sets *errno* to indicate the error.

ERRORS

The *m_locspec_to_host*( ) function may fail if:

[EINVAL]
: The **LocaleSpec** specified by *locspec* does not contain any network locale specification. This occurs when the **LocaleSpec** is created but not set by any of the functions *m_locspec_from_nettoken*( ), *m_locspec_from_netstring*( ) or *m_locspec_from_host*( ).

[ENOSYS]
: The **LocaleSpec** specified by *locspec* is not supported as a locale on the host system.

APPLICATION USAGE

Refer to the *m_createlocspec*( ) function for an example of the use of this function.

SEE ALSO

m_locspec_from_host( ), *m_createlocspec*( ), *m_destroylocspec*( ), *m_setlocale*( ), **<mlocale.h>**.

NAME

m_locspec_to_nettoken — get network locale specification token using locale specification

SYNOPSIS

```
#include <mlocale.h>

LocaleNetToken  m_locspec_to_nettoken(const LocaleSpec locspec);
```

DESCRIPTION

The *m_locspec_to_nettoken*( ) function returns the network locale specification token associated with the object *locspec* of type **LocaleSpec**. The returned token corresponds to a network locale specification associated with the *locspec* (as specified by category LC_ALL) that may be distributed over networks.

The *m_locspec_to_nettoken*( ) function is intended for low-level protocol communication services, for example, RPC run-time services that need to convert the *locspec* into a well known type. Most applications are expected to convert the locspec into an **Attrobj** locale object for processing.

RETURN VALUE

Upon successful completion, the *m_locspec_to_nettoken*( ) function returns a token of type **LocaleNetToken** representing the network locale specification. Otherwise the *m_locspec_to_nettoken*( ) function returns [NoLocaleNetToken] and sets *errno* to indicate the error.

ERRORS

The *m_locspec_to_nettoken*( ) function may fail if:

[EBADF]
: The **LocaleSpec** object is invalid.

[EINVAL]
: The **LocaleSpec** specified by *locspec* cannot be represented as a LocaleNetToken. This can occur when the **LocaleSpec** contains a network locale specification of string form but the token form cannot be found.

APPLICATION USAGE

Refer to the *m_createlocspec*( ) function for a description of the use of this function.

SEE ALSO

m_locspec_from_nettoken( ), *m_createlocspec*( ), *m_destroylocspec*( ), *m_setlocale*( ), *m_localespec_from_nettoken*( ), *m_localespec_to_nettoken*( ), **<mlocale.h>**.

NAME

m_mb_cur_max — get maximum number of bytes using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <stdlib.h>

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

DESCRIPTION

The *m_mb_cur_max*( ) function returns an integer value that is the maximum number of bytes in any character specified by the locale defined by *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_mb_cur_max*( ) function returns the maximum number of bytes in any character associated with the locale defined by *attrobj*. Otherwise, the *m_mb_cur_max*( ) function returns zero and may set *errno*.

ERRORS

The *m_mb_cur_max*( ) function may fail if:

[EBADF]
The attribute object is invalid.

SEE ALSO

mblen( ), *mbtowc*( ), *m_setlocale*( ), *wctomb*( ), **<mlocale.h>**. **<stdlib.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_nl_langinfo — get language information using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <langinfo.h>
#include <limits.h>

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

DESCRIPTION

The *m_nl_langinfo*( ) function returns a pointer to the application supplied buffer *buf* into which it places information relevant to the particular language or cultural area identified in the locale identified by *attrobj*. No more than *bufsize* bytes are placed into the array pointed to by *buf* including any null terminating byte. The size of string returned by *m_nl_langinfo*( ) (including the terminating null byte) is guaranteed to never exceed MAX_INFO_MSG_LEN, defined in **<limits.h>**. The manifest constant names and values of *item* are defined in **<langinfo.h>**, for example:

```
AttrObject portuguese, english;
m_nl_langinfo(portuguese, ABDAY_1, buf, sizeof(buf))
m_nl_langinfo(english, ABDAY_1, buf, sizeof(buf))
```

would return a pointer to the string "Dom" in the first call with a Portuguese *attrobj* and "Sun" in the second call with an English *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

In a locale where *langinfo* data is not defined, *m_nl_langinfo*( ) copies into the array pointed to by *buf* the corresponding string in the POSIX locale. In all locales, *m_nl_langinfo*( ) returns a pointer to an empty string if *item* contains an invalid string.

ERRORS

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *nl_langinfo*( ) function when called with the program's locale set to the locale identified by *attrobj*.

The returned value of type **char*** is encoded in the codeset of the locale identified by *attrobj*. If a wide-character form of the data is needed, it should be converted from coded characters form to wide-character form using the *mbsrtowcs*( ) function.

The following shows how the code set of the locale defined by *attrobj* can be determined:

```
AttrObject locale;
char *buf[MAX_INFO_MSG_LEN];
char* codeset = m_nl_langinfo( locale, CODESET, &buf, MAX_INFO_MSG_LEN);
```

SEE ALSO

m_createattrobj( ), *m_mbstowcs*( ), *m_setlocale*( ), *nl_langinfo*( ), **<langinfo.h>**, **<mlocale.h>**. **<limits.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_setlocale — set locale object

SYNOPSIS

```
#include <mlocale.h>
#include <locale.h>

char *m_setlocale(AttrObject *attrobj, const int category,
                  const char *locale);
```

DESCRIPTION

The *m_setlocale*( ) function selects the appropriate piece of the locale attribute within *attrobj*, as defined by the *category* and *locale* arguments, and may be used to change or query the entire locale attribute or portions thereof.

The *m_setlocale*( ) function selects the same locale as the *setlocale*( ) function when called with the same *category* and *locale* arguments; except that they are populated into *attrobj* rather than into the program's current locale. The *category* and *locale* arguments may be set to any value defined by the *setlocale*( ) function to get similar behaviour. Refer to the *setlocale*( ) function for a description of *category* and *locale*. Changes to the categories in the current locale do not affect the categories in the locale attribute within *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_setlocale*( ) function returns the string associated with the specified category for the new locale. Otherwise, the *m_setlocale*( ) function returns a null pointer and the locale attribute is not changed.

A null pointer for *locale* causes the *m_setlocale*( ) function to return a pointer to the string associated with the *category* for the locale attribute within *attrobj*. The locale attribute within *attrobj* is not changed.

The string returned by the *m_setlocale*( ) function is such that a subsequent call with that string and its associated *category* restores that part of the locale attribute. The string returned must not be modified by the program, but may be overwritten by a subsequent call to the *m_setlocale*( ) function.

ERRORS

The *m_setlocale*( ) function may fail if:

[EBADF]
    The attribute object is invalid.

APPLICATION USAGE

The following code illustrates how a program can initialise a locale object for two languages simultaneously:

```
AttrObject loc1;
AttrObject loc2;
loc1 = m_createattrobj();
loc2 = m_createattrobj();
m_setlocale(&loc1, LC_ALL, "de_DE");
m_setlocale(&loc2, LC_ALL, "nl_NL");
```

The following code illustrates how a program can initialise both the program's (current) locale and a separate locale object. The current locale is set according to the native environment corresponding to the *LC_** and *LANG* environment variables. While the locale object, *loc1*, is set to a specified locale "de_DE".

```
AttrObject loc1;
setlocale(LC_ALL, "");
m_setlocale(&loc1, LC_ALL, "de_DE");
```

Changing the program's current locale by means of *setlocale*( ) has no effect on any locale attributes that are already opened by calls to *m_setlocale.*( ) The following shows how the current locale can be used in a multi-locale environment:

```
AttrObject default_locale;
default_locale = m_createattrobj();
m_setlocale( &default_locale, category, setlocale(category, NULL));
```

SEE ALSO

m_createattrobj( ), **<mlocale.h>**, **<locale.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_sprintf — print formatted output to a buffer using locale object

SYNOPSIS

```
#include <stdio.h>

int m_sprintf(mbstate_t *ps, char *s, size_t n, const char *format, ..);
```

DESCRIPTION

The *m_sprintf*( ) function is equivalent to the *sprintf*( ) function, except that it uses an explicit **mbstate_t** object. The **mbstate_t** object pointed to by *ps* stores the current parse state of the stream. If the attribute object associated with the **mbstate_t** object has been associated with a locale, then all locale-dependent formatting uses that locale.

In addition, all locale-dependent behaviour (conversion, radix formatting and character classification) will be performed using the locale within the **mbstate_t** object *ps*. Any reference to the global locale in the *fprintf*( ) function definition will apply to the locale of the **mbstate_t** object. Specifically, the white space definition (see %s) shall use the multi-locale function *m_iswctype*( ) (with Space attribute).

The *m_sprintf*( ) function places output followed by the null byte ' ', in consecutive bytes starting at **s*. No more than *n* bytes are written including the null byte (unless *n* is zero). It is the user's responsibility to ensure that enough space is available.

This function converts, formats and prints its arguments under control of the format string pointed to by *format*. The *format* string is identical to that of the *fprintf*( ) function as defined by the **XSH, Issue 4** specification.

The following extends the **XSH, Issue 4** specification of the *format* string to be in agreement with the MSE standard.

Adjust the description of the qualifiers h, l and L to include the additional phrases:

''an optional l specifying that a following c conversion specifier applies to a wint_t argument; an optional l specifying that a following s conversion specifier applies to a pointer to a wchar_t argument;''

Replace the description of the c conversion specifier with:

c If no l qualifier is present, the int argument is converted to an unsigned char, and the resulting character is written. Otherwise, the wint_t argument is converted as if by an ls conversion specification with no precision and an argument that points to a two-element array of wchar_t, the first element containing the wint_t argument to the lc conversion specification and the second a null wide character.

Replace the description of the s conversion specifier with:

s If no l qualifier is present, the argument shall be a pointer to an array of character type. Characters from the array are written up to (but not including) a terminating null character. If the precision is specified, no more than that many characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null character.

If an l qualifier is present, the argument shall be a pointer to an array of wchar_t type. Wide characters from the array are converted to multi-byte characters (each as if by a call to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide character is converted) up to and including a terminating null wide character. The resulting multi-byte characters are written up to (but not including) the terminating null character (byte). If no precision is specified, the array shall contain a null wide character. If a precision is specified, no more than that many characters (bytes) are written (including shift sequences, if any), and the array shall contain a null wide character if, to equal the multi-byte character sequence length given by the precision, the function would need to access a wide character one past the end of the array. In no case is a partial multi-byte character written.

RETURN VALUE

Upon successful completion, this function returns the number of bytes transmitted excluding the terminating null, or a negative value if an output error is encountered.

ERRORS

The *m_sprintf*( ) function may fail if:

[EILSEQ]
A wide-character code that does not correspond to a valid character has been detected.

[EINVAL]
There are insufficient arguments.

[ENOMEM]
Insufficient storage space is available.

[EBADF]
The **mbstate_t** object is invalid or was not created by a call to *m_creatembstate*( ).

APPLICATION USAGE

The *m_sprintf*( ) function enables an application to produce the same result as a call to *fprintf*( ) on a stream which has been associated with an attribute object by a call to *m_fattr*( ).

SEE ALSO

m_swprintf( ), *m_sscanf*( ), *m_swscanf*( ).

CHANGE HISTORY

New for the DIS, with input from the MSE standard.

NAME

m_sscanf — convert formatted input into an array using locale object

SYNOPSIS

```
#include <stdio.h>

int m_sscanf(mbstate_t *ps, const char *s, const char *format, ... );
```

DESCRIPTION

The *m_sscanf*( ) function is equivalent to the *sscanf*( ) function, except that it uses an explicit **mbstate_t** object. The **mbstate_t** object pointed to by *ps* stores the current parse state of the stream. If the attribute object associated with the **mbstate_t** object has been associated with a locale, then all locale-dependent formatting uses that locale.

In addition, all locale-dependent behaviour (conversion, radix formatting and character classification) will be performed using the locale within the **mbstate_t** object *ps*. Any reference to the global locale in the *fprintf*( ) function definition will apply to the locale of the **mbstate_t** object. Specifically, the white space definition (see %s) shall use the multi-locale function *m_iswctype*( ) (with Space attribute).

The *sscanf*( ) function is equivalent to the *fscanf*( ) function, except that it reads from the character array pointed to by *s*, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the *fwscanf*( ) function.

The *format* string is identical to that of the *fprintf*( ) function as modified by the **XSH, Issue 4** specification.

The following extends the **XSH, Issue 4** specification definition of the *format* string to be in agreement with the MSE standard.

Adjust the description of the qualifiers h, l and L to include the additional sentences:

‘‘The conversion specifiers c, s and [shall be preceded by l if the corresponding argument is a pointer to wchar_t rather than a pointer to a character type.’’

Replace the definition of directive failure (page 135, lines 34-36, beginning with, "If the length of the input item is zero...") with:

‘‘If the length of the input item is zero, the execution of the directive fails; this condition is a matching failure unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.’’

Replace the description of the s conversion specifier with:

s Matches a sequence of non-white-space characters. 16) If no l qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an l qualifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multibyte character is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first multi-byte character is converted. The corresponding argument shall be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Replace the first two sentences of the description of the [conversion specifier with:

[Matches a non-empty sequence of characters from a set of expected characters (the scanset). If no l qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an l qualifier is present, the input shall be a sequence of multibyte characters that begins in the initial shift state. Each multi-byte character is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first multi-byte character is converted. The corresponding argument shall be a pointer to an array of wchar_t large enough to accept the sequence and the terminating null wide character, which will be added automatically.

Replace the description of the c conversion specifier with:

c Matches a sequence of characters of the number specified by the field width (1 if no field width is present in the directive). If no l qualifier is present, the corresponding argument shall be a pointer to a character array large enough to accept the sequence. No null character is added.

If an l qualifier is present, the input shall be a sequence of multi-byte characters that begins in the initial shift state. Each multi-byte character in the sequence is converted to a wide character as if by a call to the *mbrtowc*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first multi-byte character is converted. The corresponding argument shall be a pointer to the initial element of an array of wchar_t large enough to accept the resulting sequence of wide characters. No null wide character is added.

The above extension is applicable to all the formatted input functions specified in the **XSH, Issue 4** specification.

RETURN VALUE

The function returns the macro EOF if an input failure occurs before any conversion. Otherwise, the *m_sscanf*( ) function returns the number of input items assigned, which can be fewer than provided for, or even zero in the event of an early matching failure.

ERRORS

The *m_sscanf*( ) function may fail if:

[EBADF]
The **mbstate_t** object is invalid or was not created by a call to *m_creatembstate*( ).

[EILSEQ]
Input byte sequence does not form a valid character.

[EINVAL]
There are insufficient arguments.

APPLICATION USAGE

The *m_sscanf*( ) function enables an application to produce the same result as a call to *fscanf*( ) on a stream which has been associated with an attribute object by a call to *m_fattr*( ).

SEE ALSO

m_sprintf( ), *m_swprintf*( ), *m_swscanf*( ).

CHANGE HISTORY

New for the DIS, with input from the MSE standard.

NAME

m_strcoll — string comparison using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

int m_strcoll(const AttrObject attrobj, const char *s1,
              const char *s2);
```

DESCRIPTION

The *m_strcoll*( ) function compares the string pointed to by *s1* to the string pointed to by *s2*, both interpreted as appropriate to the LC_COLLATE category of the locale defined in *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_strcoll*( ) function returns an integer greater than, equal to or less than zero, according to whether the string pointed to by *s1* is greater than, equal to or less than the string pointed to by *s2* when both are interpreted as appropriate to the locale defined in *attrobj*. On error, *m_strcoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *m_strcoll*( ) function may fail if:

[EBADF]
    The attribute object is invalid.

[EINVAL]
    The *s1* or *s2* arguments contain characters outside the domain of the collating sequence.

APPLICATION USAGE

This function behaves in the same manner as the *strcoll*( ) function when called with the current locale set to the locale defined in *attrobj*.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

The *m_strfxrm*( ) and *strcmp*( ) functions should be used when doing many, repeated comparisons of the same strings, such as sorting large lists.

SEE ALSO

m_strxfrm, *m_wcscoll*( ), *m_wcsxfrm*, *strcoll*( ), *strcmp*( ), **<mlocale.h>, <string.h>**

CHANGE HISTORY

Derived from Version 1 of this document.

NAME

m_strcspn — get length of complementary substring

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

size_t m_strcspn(const AttrObject attrobj, const char *str1,
                 const char *str2);
```

DESCRIPTION

The *m_strcspn*( ) function computes the length (in bytes) of the maximum initial segment of the string of coded characters pointed to by *str1* that consists entirely of coded characters *not* from the string pointed to by *str2*. Comparisons are performed only on complete coded characters and at coded character boundaries. If the locale defined by *attrobj* is defined with possible composite sequences, then both *str1* and *str2* may contain composite sequences. In the case of composite sequences occurring in either string, comparisons for matching are performed only on complete composite sequences.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_strcspn*( ) function returns the length (in bytes) of the segment; no return value is reserved to indicate an error.

ERRORS

The *m_strcspn*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *strcspn*( ) function when called with the current locale set to the locale defined by *attrobj*. The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

It is the caller's responsibility to ensure the *str1* and *str2* are pointed to the beginning of the coded character boundary; otherwise, the result might be undefined.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

SEE ALSO

m_wcsspn( ), *m_strspn*( ), *wcscspn*( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strerror — get error message string using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

char *m_strerror(const AttrObject attrobj, int errnum);
```

DESCRIPTION

The *m_strerror*( ) function maps the error number in *errnum* to a locale-dependent error message string and returns a pointer thereto. The string pointed to must not be modified by the program, but may be overwritten by a subsequent call to *m_strerror*( ) with the locale identified by *attrobj*.

The contents of the error message strings returned by *m_strerror*( ) should be determined by the setting of the LC_MESSAGES category in the locale identified by *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, *m_strerror*( ) returns a pointer to the generated message string. On error *errno* may be set, but no return value is reserved to indicate an error.

ERRORS

The *m_strerror*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

[EINVAL]
: The value of *errnum* is not a valid error message number.

APPLICATION USAGE

This function behaves in the same manner as the *strerror*( ) function when called with the current locale set to the locale identified by *attrobj*.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to zero, then call *m_strerror*( ), then check *errno* and if it is non-zero, assume an error has occurred.

This function returns a pointer to a string located in global process address space. This requires applications that desire thread-specific data to manage concurrent access to the returned string.

SEE ALSO

strerror( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_strfmon — convert monetary value to string using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <monetary.h>

ssize_t m_strfmon(const AttrObject attrobj, char *s, size_t maxsize,
                  const char *format, ...);
```

DESCRIPTION

The *m_strfmon*( ) function places characters into the array pointed to by *s* as controlled by the string pointed to by *format*. No more than *maxsize* bytes are placed into the array.

The format is a character string that contains two types of objects: plain characters, which are simply copied to the output stream, and conversion specifications, each of which results in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

This function behaves in the same manner as the *strfmon*( ) function when called with the current locale set to the locale identified by *attrobj*. Refer to the *strfmon*( ) function for a description of the conversion specification.

Locale Information

The LC_MONETARY category of the locale identified by *attrobj* affects the behaviour of this function including the monetary radix character (which may be different from the numeric radix character affected by the LC_NUMERIC category), the grouping separator, the currency symbols and formats. The international currency symbol should be conformant with the ISO 4217 standard.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, the *m_strfmon*( ) function returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, –1 is returned, the contents of the array are indeterminate and *errno* is set to indicate the error.

ERRORS

The *m_strfmon*( ) function fails if:

[E2BIG]
: Conversion stopped due to lack of space in the buffer.

The *m_strfmon*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

EXAMPLES

Refer to the *strfmon*( ) for examples of different formats.

SEE ALSO

m_localeconv( ), *localeconv*( ), *strfmon*( ), **<mlocale.h>**, **<monetary.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_strftime — convert date and time to string using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <time.h>

size_t m_strftime(const AttrObject attrobj, char *s, size_t maxsize,
                  const char *format, const struct tm *timptr);
```

DESCRIPTION

The *m_strftime*( ) function places bytes into the array pointed to by *s* as controlled by the string pointed to by *format*. The *format* string consists of zero or more conversion specifications and ordinary characters. A conversion specification consists of a % character and a terminating conversion character that determines the conversion specification's behaviour. All ordinary characters (including the terminating null byte) are copied unchanged into the array. If copying takes place between objects that overlap, the behaviour is undefined. No more than *maxsize* bytes are placed into the array. Each conversion specification is replaced by appropriate characters as described in the *strftime* function. The appropriate characters are determined by the locale identified by *attrob* and by the values contained in the structure pointed to by *timptr*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If the total number of resulting bytes including the terminating null byte is not more than *maxsize*, the *m_strftime*( ) function returns the number of bytes placed into the array pointed to by *s*, not including the terminating null byte. Otherwise, zero is returned and the contents of the array are indeterminate.

ERRORS

The *m_strftime*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as if the *strftime*( ) function were called with the current locale set to the locale identified by *attrobj*. Refer to the *strftime*( ) function for a description of the conversion specification and application usage tips.

SEE ALSO

strftime( ), *asctime*( ), *clock*( ), *ctime*( ), *difftime*( ), *gmtime*( ), *localtime*( ), *m_strptime*( ), *mktime*( ), *strptime*( ), *time*( ), *utime*( ), **<mlocale.h>**, **<time.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_strpbrk — scan text object for one or more coded characters

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

char *m_strpbrk(const AttrObject attrobj, const char *str1,
                   const char *str2);
```

DESCRIPTION

The *m_strpbrk*( ) function locates the first occurrence of any coded character in the string pointed to by *str1* of any coded characters from the string pointed to by *str2*. Comparisons are performed only on complete coded characters and at coded character boundaries.

If the locale defined by *attrobj* is defined with possible composite sequences, then both *str1* and *str2* may contain composite sequences. In the case of composite sequences occurring in either string, comparisons for matching are performed only on complete composite sequences.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

On successful completion, *m_strpbrk*( ) returns a pointer to a coded character in *str1* or a null pointer if no coded character from *str2* occurs in *str1*.

ERRORS

The *m_strpbrk*( ) function may fail if:

[EBADF]
  The attribute object is invalid.

APPLICATION USAGE

The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

SEE ALSO

m_setlocale( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strptime — date and time conversion using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <time.h>

char *m_strptime(const AttrObject attrobj, const char *buf,
                 const char *format, struct tm *tm);
```

DESCRIPTION

The *m_strptime*( ) function converts the character string pointed to by *buf* to values which are stored in the **tm** structure pointed to by *tm*, using the format specified by *format*.

This function behaves in the same manner as the *strptime*( ) function when called with the current locale set to the locale identified by *attrobj*. Refer to the *strptime*( ) function for a description of the format specification.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, *m_strptime*( ) returns a pointer to the character following the last character parsed. Otherwise, a null pointer is returned.

ERRORS

The *m_strptime*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

The LC_TIME category of the locale identified by *attrobj* affects the behaviour of this function when references are made to locale-dependent information in the *format* specification.

SEE ALSO

m_strftime( ), *scanf*( ), *strftime*( ), *strptime*( ), *time*( ), **<time.h>**, **<mlocale.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_strscanfor — scans a coded character string for a specific character

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

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

DESCRIPTION

The *m_strscanfor*( ) function scans the coded character string pointed to by *s* at the offset specified by *position* for the first coded character that matches or does not match the set of classification criteria specified by *condition*.

The argument *condition* specifies a set of classification criteria that can be ORed bitwise. The following conditions are reserved for the standard classification criteria and are defined in the **APPLICATION USAGE** section of the *m_wcsscanfor*( ) function.

```
typedef enum { Alphabetic, WhiteSpace, Control, Digit, Graphic,
               Lowercase, Uppercase, Printing, Punctuation, HexDigit,
               LineBreakCharacter, LineBreakHyphen, LineBreakScript,
               WordBoundary, SentenceBoundary, ParagraphBoundary,
               CharsetBoundary, ScriptBoundary, CompositeBoundary
             } ScanCondition;
```

The argument *inverse* determines the rules for scanning, and is an integer type containing one of the values defined for the following enumeration type:

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

If *inverse* is set to False, the function searches for the first coded character that matches the specified condition; if *inverse* is set to True, the function searches for the first coded character that does not match the condition.

The argument *direction* determines the direction in which the scanning takes place. It is an integer type containing one of the values defined for the following enumeration type:

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

If *direction* is set to Forw, the function scans from *position* to the end of the coded character string *s*; if *direction* is set to Back, the function scans from *position* to the beginning of the coded character string *s*.

The rules of any classification criteria are determined by the locale defined by *attrobj*.

The search begins from the coded character identified by *s*, and continues through the next *num_bytes* bytes.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If all the criteria specified by *condition* are met, a non-negative integer identifying the location of the first coded character meeting the criteria is returned. The non-negative integer value returned indicates the offset number of bytes from the coded character identified by *s*, to the coded character at which the condition is satisfied. If no coded character meets the specified criteria, a (**size_t**)–1 is returned and *errno* is set to indicate the error.

ERRORS

The *m_strscanfor*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The *condition* is invalid.

[EILSEQ] No *condition* could be found in the supplied coded character string.

APPLICATION USAGE

It is recommend that the *position* value be updated on subsequent calls since certain languages (for example, Thai) may require viewing the code elements that precede the *position* point (when direction = Forw).

Refer to the *m_wcsscanfor*( ) function for a description of the classification criteria.

SEE ALSO

m_wcsscanfor( ), *m_setlocale*( ), *m_iswctype*( ), **<mlocale.h>**, **<string.h>**.

NAME

m_strspn — get length of substring using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

size_t m_strspn(const AttrObject attrobj, const char *str1,
                const char *str2);
```

DESCRIPTION

The *m_strspn*( ) function computes the length (number of bytes) of the maximum initial segment of the character string pointed to by *str1* which consists entirely of coded characters from the string pointed to by *str2*. Comparisons are performed only on complete coded characters and at coded character boundaries.

If the locale defined by *attrobj* is defined with possible composite sequences, then both *str1* and *str2* may contain composite sequences. In the case of composite sequences occurring in either string, comparisons for matching are performed only on complete composite sequences.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_strspn*( ) function returns the length (in bytes) of the segment; no return value is reserved to indicate an error.

ERRORS

The *m_strspn*( ) function may fail if:

[EBADF]
    The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *strspn*( ) function when called with the current locale set to the locale defined by *attrobj*. The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

SEE ALSO

wcscspn( ), *m_wcscspn*( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strstr — find substring

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

char *m_strstr(const AttrObject attrobj, const char *str1,
               const char *str2);
```

DESCRIPTION

The *m_strstr*( ) function locates the first occurrence in the string of coded characters pointed to by *str1* of the sequence of coded characters (excluding the terminating null character code) in the string pointed to by *str2*. Comparisons are performed only on complete coded characters and at coded character boundaries.

If the locale defined by *attrobj* is defined with possible composite sequences, then both *str1* and *str2* may contain composite sequences. In the case of composite sequences occurring in either string, comparisons for matching are performed only on complete composite sequences.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_strstr*( ) function returns a pointer to the located string or a null pointer if the string of coded characters is not found.

If *str2* points to a character string with zero length, the function returns *str1*.

ERRORS

The *m_strstr*( ) function may fail if:

[EBADF]
  The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *wcswcs*( ) function when called with the current locale set to the locale defined by *attrobj*, except that multi-byte strings are processed instead of wide-character strings. The locale defined in *attrobj* may be used to match characters in cases where different sequences of combining characters may be used to define a character.

SEE ALSO

wcschr( ), *wcswcs*( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strtod — convert string to double-precision number using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <stdlib.h>

double m_strtod(const AttrObject attrobj, const char *str,
                char **endptr);
```

DESCRIPTION

The *m_strtod*( ) function converts the initial portion of the string pointed to by *str* to type **double** representation.

The radix character is defined in the locale (category LC_NUMERIC) identified by *attrobj*.

Refer to the *strtod*( ) function for a description of the expected form of the input string.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_strtod*( ) function returns the converted value. If no conversion could be performed, zero is returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, ±HUGE_VAL is returned (according to the sign of the value) and *errno* is set to [ERANGE].

If the correct value would cause an underflow, zero is returned and *errno* is set to [ERANGE].

ERRORS

The *m_strtod*( ) function fails if:

[ERANGE]
The value to be returned would cause overflow or underflow.

The *m_strtod*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
No conversion could be performed.

APPLICATION USAGE

This function behaves in the same manner as the *strtod*( ) function when called with the current locale set to the locale defined by *attrobj*.

Because zero is returned on error and is also a valid return on success, an application wishing to check for error situations should set *errno* to 0, then call *m_strtod*( ), then check *errno* and if it is non-zero, assume an error has occurred.

SEE ALSO

isspace( ), *iswctype*( ), *localeconv*( ), *m_localeconv*( ), *m_isctype*( ), *m_setlocale*( ), *m_strtol*( ), *m_wcstod*( ), *m_iswctype*( ), *scanf*( ), *strtol*( ), *wcstod*( ), *wcstol*( ), *wctype*( ), **<mlocale.h>**, **<stdlib.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_strtok — split string into tokens using a text context object

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

char *m_strtok(char *s1, const char *s2, mbstate_t *ps);
```

DESCRIPTION

A sequence of calls to the *m_strtok*( ) function breaks the string pointed to by *s1* into a sequence of tokens, each of which is delimited by a coded character from the string pointed to by *s2*. The *ps* argument points to a text context object of type **mbstate_t** that is used to store any tokenising context necessary for it to continue scanning the same string.

For the first call in the sequence, *s1* points to a coded character string, while in subsequent calls for the same string, *s1* shall be a null pointer. If *s1* is a null pointer, the value pointed to by *ps* shall match that stored by the previous call for the same string. The separator string pointed to by *s2* may be different from call to call.

The first call in the sequence searches the string pointed to by *s1* for the first coded character that is not contained in the current separator string pointed to by *s2*. If no such coded character is found, then there are no tokens in the string pointed to by *s1* and the *m_strtok*( ) function returns a null pointer. If such a coded character is found, it is the start of the first token.

The *m_strtok*( ) function then searches from there for a coded character that *is* contained in the current separator string. If no such coded character is found, the current token extends to the end of the string pointed to by *s1*, and subsequent searches in the same string for a token returns a null pointer. If such a coded character is found, it is overwritten by a null coded character, which terminates the current token.

In all cases, the *m_strtok*( ) function stores sufficient information in the **mbstate_t** pointed to by *ps* so that subsequent calls, with a null pointer for *s1*, shall start searching just past the element overwritten by a null coded character (if any).

If the locale defined by *ps* is defined with possible composite sequences, both *s1* and *s2* may contain composite sequences. If composite sequences are contained in *s2*, the entire composite sequence is used to delimit tokens. It is implementation dependent how composite sequence matching is performed.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_strtok*( ) function returns a pointer to the first coded character of a token. Otherwise, if there is no token, *m_strtok*( ) returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

This function behaves in the same manner as the *strtok*( ) function when called with the current locale set to the locale defined by *attrobj*.

The encoding of coded characters is defined by the locale associated with *ps*. An example of searching for tokens in a multi-locale environment is as follows:

```
#include <mlocale.h>

static char_t str1[] = "?a???b,,,#c";
static char_t str2[] = " ";
char_t *t;
AttrObject foo_locale;            /* previously initialised */
AttrObject C_locale;              /* previously initialised */
mbstate_t ps1 = m_creatembstate(foo_locale );
mbstate_t ps2 = m_creatembstate(C_locale );

t = m_strtok(str1, "?", &ps1);    /* t points to the token "a" */
t = m_strtok(NULL, ",", &ps1);    /* t points to the token "??b" */
t = m_strtok(str2, " \t", &ps2);  /* t is a null pointer */
t = m_strtok(NULL, "#", &ps1);    /* t points to the token ",," */
t = m_strtok(NULL, "?", &ps2);    /* t is a null pointer */
t = m_strtok(NULL, "?", &ps1);    /* t is a null pointer */

m_destroymbstate(ps1);
m_destroymbstate(ps2);
```

SEE ALSO

m_creatembstate( ), *m_setlocale*( ), **<mlocale.h>**, **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strtol — convert string to long integer using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <stdlib.h>

long int m_strtol(const AttrObject attrobj, const char *str,
                  char **endptr, int base);
```

DESCRIPTION

The *m_strtol*( ) function converts the initial portion of the string pointed to by *str* to a type **long int** representation.

This function behaves in the same manner as the *strtol*( ) function when called with the current locale set to the locale identified by *attrobj*. Refer to the *strtol*( ) function for a description of the expected form of the input string.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion the *m_strtol*( ) function returns the converted value, if any. If no conversion could be performed 0 is returned and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG_MAX} or {LONG_MIN} is returned (according to the sign of the value) and *errno* is set to [ERANGE].

ERRORS

The *m_strtol*( ) function fails if:

[ERANGE]
: The value to be returned is not representable.

The *m_strtol*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

[EINVAL]
: The value of *base* is not supported.

APPLICATION USAGE

Because 0, {LONG_MIN} and {LONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *m_strtol*( ), then check *errno*, and if it is non-zero, assume an error has occurred.

SEE ALSO

isalpha( ), *m_isctype*( ), *m_iswctype*( ), *m_strtod*( ), *m_wcstod*( ), *scanf*( ), *strtod*( ), *wcstod*( ), *wctype*( ), **<mlocale.h>**, **<stdlib.h>**.

CHANGE HISTORY

Derived from Version 1 of this document.

NAME

m_strtoul — convert string to unsigned long using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <stdlib.h>

unsigned long int m_strtoul(const AttrObject attrobj, const char *str,
                            char **endptr, int base);
```

DESCRIPTION

The *m_strtoul*( ) function converts the initial portion of the string pointed to by *str* to a type **unsigned long int** representation.

This function behaves in the same manner as the *strtoul*( ) function when called with the current locale set to the locale identified by *attrobj*. Refer to the *strtoul*( ) function for a description of the expected form of the input string.

In other than the POSIX locale, additional implementation-dependent subject sequence forms may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *str* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion the *m_strtoul*( ) function returns the converted value, if any. If no conversion could be performed, zero is returned and *errno* may be set to [EINVAL]. If the correct value is outside the range of representable values, {ULONG_MAX} is returned and *errno* is set to [ERANGE].

ERRORS

The *m_strtoul*( ) function fails if:

[ERANGE]
: The value to be returned is not representable.

The *m_strtoul*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

[EINVAL]
: The value of *base* is not supported.

APPLICATION USAGE

Because zero and {ULONG_MAX} are returned on error and are also valid returns on success, an application wishing to check for error situations should set *errno* to 0, then call *m_strtoul*( ), then check *errno* and if it is non-zero, assume an error has occurred.

Unlike *m_strtod*( ) and *m_strtol*( ), *m_strtoul*( ) always returns a non-negative number; therefore, using the return value of *m_strtoul*( ) for out-of-range numbers with *m_strtoul*( ) could cause more severe problems than just loss of precision if those numbers can ever be negative.

SEE ALSO

isalpha( ), *iswctype*( ), *m_isctype*( ), *m_iswctype*( ), *m_strtod*( ), *m_strtol*( ), *m_wcstoul*( ), *scanf*( ), *strtod*( ), *strtol*( ), *wcstoul*( ), *wctype*( ), **<mlocale.h>**, **<stdlib.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_strxfrm — string transformation using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <string.h>

size_t m_strxfrm(const AttrObject attrobj, char *s1,
                 const char *s2, size_t n);
```

DESCRIPTION

The *m_strxfrm*( ) function transforms the string pointed to by *s2* and places the resulting string into the array pointed to by *s1*. The transformation is such that if the *strcmp*( ) function is applied to two transformed strings, it returns a value greater than, equal to or less than 0, corresponding to the result of the *m_strcoll*( ) function applied to the same two original strings with the same locale identified by *attrobj*. No more than *n* bytes are placed into the resulting array pointed to by *s1*, including the terminating null byte. If *n* is zero, *s1* is permitted to be a null pointer. If copying takes place between objects that overlap, the behaviour is undefined.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_strxfrm*( ) function returns the length of the transformed string (not including the terminating null byte). If the value returned is *n* or more, the contents of the array pointed to by *s1* are indeterminate.

On error, the *m_strxfm*( ) function returns (**size_t**)−1 and sets *errno* to indicate the error.

ERRORS

The *m_strxfrm*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The string pointed to by the *s2* argument contains characters outside the domain of the collating sequence.

APPLICATION USAGE

The transformation function is such that two transformed strings can be ordered by the *strcmp*( ) function as appropriate to collating sequence information in the locale (category LC_COLLATE) identified by *attrobj*.

This function behaves in the same manner as the *strxfrm*( ) function when called with the current locale set to the locale identified by *attrobj*.

The fact that when *n* is zero, *s1* is permitted to be a null pointer, is useful to determine the size of the *s1* array prior to making the transformation.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strxfrm*( ), then check *errno*, and if it is non-zero, assume an error has occurred.

SEE ALSO

strcmp( ), *m_strcoll*( ), *m_wcscoll*( ), *strcoll*( ), *strxfrm*( ), **<mlocale.h>**. **<string.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_swprintf — print formatted output to a buffer using locale object

SYNOPSIS

```
#include <stdio.h>

int m_swprintf(mbstate_t *ps, wchar_t *s, size_t n,
        const wchar_t *format, ..);
```

DESCRIPTION

The *m_swprintf*( ) function is equivalent to the *swprintf*( ) function specified in the MSE standard except that it uses an explicit **mbstate_t** object pointed to by *ps*, rather than an **mbstate_t** object associated with the stream by a call to *m_fattr*( ). The **mbstate_t** object pointed to by *ps* stores the current parse state of the stream. If the attribute object associated with the **mbstate_t** object has been associated with a locale, then all locale-dependent formatting uses that locale.

In addition, all locale-dependent behaviour (conversion, radix formatting and character classification) will be performed using the locale within the **mbstate_t** object *ps*. Any reference to the global locale in the *fprintf*( ) function definition will apply to the locale of the **mbstate_t** object. Specifically, the white space definition (see %s) shall use the multi-locale function *m_iswctype*( ) (with Space attribute).

The *m_swprintfi*( ) function writes output to the array of wide characters pointed to by *s*, under control of the wide string pointed to by format that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the *format*, the behaviour is undefined. If the *format* is exhausted while arguments remain, the excess arguments are evaluated (as always) but are otherwise ignored. The *fwprintf*( ) function returns when the end of the format string is encountered.

The *format* string is identical to that of the *m_sprintf*( ) function and is extended to be in agreement with the MSE standard of being represented using the wchar_t encoding.

The format is composed of zero or more directives: ordinary wide characters (not %), and conversion specifications. The processing of conversion specifications is as if they were replaced in the format string by wide-character strings that are each the result of fetching zero or more subsequent arguments and converting them, if applicable, according to the corresponding conversion specifier. The expanded wide-character format string is then written to the output stream.

Each conversion specification is introduced by the wide character %. After the %, the following appear in sequence:

- Zero or more flags (in any order) that modify the meaning of the conversion specification.
- An optional minimum field width . If the converted value has fewer wide characters than the field width, it is padded with spaces (by default) on the left (or right, if the left adjustment flag, described later, has been given) to the field width. The field width takes the form of an asterisk * (described later) or a decimal integer.
- An optional precision that gives the minimum number of digits to appear for the d, i, o, u, x and X conversions, the number of digits to appear after the decimal-point character for e, E and f conversions, the maximum number of significant digits for the g and G conversions, or the maximum number of wide characters to be written from a string in s conversion. The precision takes the form of a period (.) followed either by an asterisk * (described later) or by an optional decimal integer; if only the period is specified, the precision is taken as zero. If a precision appears with any other conversion specifier, the behaviour is undefined.

- An optional l (ell) specifying that a following c conversion specifier applies to a wint_t argument; an optional l specifying that a following s conversion specifier applies to a pointer to a wchar_t argument; an optional h specifying that a following d, i, o, u, x or X conversion specifier applies to a **short**int" or **unsigned short int** argument (the argument is promoted according to the integral promotions, and its value is converted to **short int** or **unsigned short int** before printing); an optional h specifying that a following n conversion specifier applies to a pointer to a **short int** argument; an optional l specifying that a following d, i , o, u, x or X conversion specifier applies to a **long int** or **unsigned long int** argument; an optional l specifying that a following n conversion specifier applies to a pointer to a **long int** argument; or an optional L specifying that a following e, E, f, g or G conversion specifier applies to a **long double** argument. If an h, l, or L appears with any other conversion specifier, the behaviour is undefined.
- A wide character that specifies the type of conversion to be applied.

As noted above, a field width, or precision, or both, may be indicated by an asterisk. In this case, an **int** argument supplies the field width or precision. The arguments specifying field width, or precision, or both, shall appear (in that order) before the argument (if any) to be converted. A negative field width argument is taken as a – flag followed by a positive field width. A negative precision argument is taken as if the precision were omitted.

The flag wide characters and their meanings are:

–	The result of the conversion is left-justified within the field. (It is right-justified if this flag is not specified.)
+	The result of a signed conversion always begins with a plus or minus sign. (It begins with a sign only when a negative value is converted if this flag is not specified.)
space	If the first wide character of a signed conversion is not a sign, or if a signed conversion results in no wide characters, a space is prefixed to the result. If the space and + flags both appear, the space flag is ignored.
#	The result is to be converted to an ‘‘alternate form’’. For o conversion, it increases the precision to force the first digit of the result to be a zero, if necessary. For x (or X) conversion, a non-zero result has 0x (or 0X) prefixed to it. For e, E, f, g and G conversions, the result always contains a decimal-point wide character, even if no digits follow it. (Normally, a decimal-point wide character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros are not be removed from the result. For other conversions, the behaviour is undefined.
0	For d, i, o, u, x, X, e, E, f, g and G conversions, leading zeros (following any indication of sign or base) are used to pad to the field width; no space padding is performed. If the 0 and – flags both appear, the 0 flag is ignored. For d, i, o, u, x and X conversions, if a precision is specified, the 0 flag is ignored. For other conversions, the behaviour is undefined.

The conversion specifiers and their meanings are:

d,i	The int argument is converted to signed decimal in the style [-]dddd . The precision specifies the minimum number of digits to appear; if the value being converted can be represented in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.
o,u,x,X	The **unsigned int** argument is converted to unsigned octal (o), unsigned decimal (u), or unsigned hexadecimal notation (x or X) in the style dddd ; the letters abcdef are used for x conversion and the letters ABCDEF for X conversion. The precision specifies the minimum number of digits to appear; if the value being converted can be represented

in fewer digits, it is expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no wide characters.

f The double argument is converted to decimal notation in the style [-]ddd.ddd, where the number of digits after the decimal-point wide character is equal to the precision specification. If the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point wide character appears. If a decimal-point wide character appears, at least one digit appears before it. The value is rounded to the appropriate number of digits.

e,E The double argument is converted in the style [-]d.ddde 1 dd, where there is one digit before the decimal-point wide character (which is non-zero if the argument is non-zero) and the number of digits after it is equal to the precision; if the precision is missing, it is taken as 6; if the precision is zero and the # flag is not specified, no decimal-point wide character appears. The value is rounded to the appropriate number of digits. The E conversion specifier produces a number with E instead of e introducing the exponent. The exponent always contains at least two digits. If the value is zero, the exponent is zero.

g,G The double argument is converted in style f or e (or in style E in the case of a G conversion specifier), with the precision specifying the number of significant digits. If the precision is zero, it is taken as 1. The style used depends on the value converted; style e (or E) is used only if the exponent resulting from such a conversion is less than –4 or greater than or equal to the precision. Trailing zeros are removed from the fractional portion of the result; a decimal-point wide character appears only if it is followed by a digit.

c If no l qualifier is present, the **int** argument is converted to a wide character as if by calling *btowc*( ) and the resulting wide character is written. Otherwise, the wint_t argument is converted to wchar_t and written.

s If no l qualifier is present, the argument shall be a pointer to a character array containing a multibyte sequence beginning in the initial shift state. Characters from the array are converted as if by repeated calls to the mbrtowc function, with the conversion state described by the **mbstate_t** object pointed to by *ps*, and written up to (but not including) the terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the converted array, the converted array shall contain a null wide character.

If an l qualifier is present, the argument shall be a pointer to an array of wchar_t type. Wide characters from the array are written up to (but not including) a terminating null wide character. If the precision is specified, no more than that many wide characters are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null wide character.

p The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable wide characters, in an implementation-defined manner.

n The argument shall be a pointer to an integer into which is written the number of wide characters written to the output stream so far by this call to *fwprintf*( ). No argument is converted.

% A % wide character is written. No argument is converted. The complete conversion specification shall be %%.

If a conversion specification is invalid, the behaviour is undefined.

If any argument is, or points to, a union or an aggregate (except for an array of char type using %s conversion, an array of wchar_t type using %ls conversion, or a pointer using %p conversion), the behaviour is undefined.

In no case does a non-existent or small field width cause truncation of a field; if the result of a conversion is wider than the field width, the field is expanded to contain the conversion result.

RETURN VALUE

The *m_swprintf*( ) function returns the number of wide characters transmitted, or a negative value if an output error occurred.

ERRORS

The *m_swprintf*( ) function may fail if:

[EILSEQ]
A wide-character code that does not correspond to a valid character has been detected.

[ENOMEM]
Insufficient storage space is available.

[EBADF]
The **mbstate_t** object is invalid or was not created by a call to *m_creatembstate*( ).

APPLICATION USAGE

The *m_swprintf*( ) function enables an application to produce the same result as a call to *fwprintf*( ) on a stream which has been associated with an attribute object by a call to *m_fattr*( ).

SEE ALSO

m_sprintf( ), *m_sscanf*( ), *m_swscanf*( ).

CHANGE HISTORY

New for the DIS, with input from the MSE standard.

NAME

m_sscanf — convert formatted input into an wide character array using locale object

SYNOPSIS

```
#include <stdio.h>

int m_swscanf(mbstate_t *ps, const wchar_t *s,
        const wchar_t *format, ... );
```

DESCRIPTION

The *m_swscanf*( ) function is equivalent to the *swscanf*( ) function in the MSE standard except that it uses an explicit **mbstate_t** object. The **mbstate_t** object pointed to by *ps* stores the current parse state of the stream. If the attribute object associated with the **mbstate_t** object has been associated with a locale, then all locale-dependent formatting uses that locale.

The *swscanf*( ) function is equivalent to the *fwscanf*( ) function except that it reads from the character array pointed to by *s*, rather than from a stream. Reaching the end of the string is equivalent to encountering end-of-file for the *fwscanf*( ) function.

In addition, all locale-dependent behaviour (conversion, radix formatting and character classification) will be performed using the locale within the **mbstate_t** object *ps*. Any reference to the global locale in the *fprintf*( ) function definition will apply to the locale of the **mbstate_t** object. Specifically, the white space definition (see %s) shall use the multi-locale function *m_iswctype*( ) (with Space attribute).

The *format* string is identical to that of the *m_sscanf*( ) function and is extended to be in agreement with the MSE standard of being represented using the wchar_t encoding.

The format is composed of zero or more directives: one or more white-space wide characters; an ordinary wide character (neither % nor a white-space wide character); or a conversion specification. Each conversion specification is introduced by a %. After the %, the following appear in sequence:

- An optional assignment-suppressing wide character *.
- An optional non-zero decimal integer that specifies the maximum field width (in wide characters).
- An optional h, l (ell) or L indicating the size of the receiving object. The conversion specifiers c, s and [shall be preceded by l if the corresponding argument is a pointer to wchar_t rather than a pointer to a character type. The conversion specifiers d, i and n shall be preceded by h if the corresponding argument is a pointer to **short int** rather than a pointer to **int**, or by l if it is a pointer to **long int**. Similarly, the conversion specifiers o, u and x shall be preceded by h if the corresponding argument is a pointer to **unsigned short int** rather than a pointer to **unsigned int**, or by l if it is a pointer to **unsigned long int**. Finally, the conversion specifiers e, f and g shall be preceded by l if the corresponding argument is a pointer to double rather than a pointer to float, or by L if it is a pointer to **long double**. If an h, l or L appears with any other conversion specifier, the behaviour is undefined.
- A wide character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The *m_swscanf*( ) function executes each directive of the format in turn. If a directive fails, as detailed below, the *m_swscanf*( ) function returns. Failures are described as input failures (if an encoding error occurs or due to the unavailability of input characters), or matching failures (due to inappropriate input).

A directive composed of white-space wide character(s) is executed by reading input up to the first non-white-space wide character (which remains unread), or until no more wide characters can be read.

A directive that is an ordinary wide character is executed by reading the next wide character of the stream. If the wide character differs from the directive, the directive fails, and the differing and subsequent wide characters remain unread.

A directive that is a conversion specification defines a set of matching input sequences, as described below for each specifier. A conversion specification is executed in the following steps.

Input white-space wide characters (as specified by the *iswspace*( ) function) are skipped, unless the specification includes a c or n specifier.

An input item is read from the stream, unless the specification includes an n specifier. An input item is defined as the longest sequence of input wide characters, not exceeding any specified field width, which is, or is a prefix of, a matching sequence. The first wide character, if any, after the input item remains unread. If the length of the input item is zero, the execution of the directive fails: this condition is a matching failure, unless end-of-file, an encoding error, or a read error prevented input from the stream, in which case it is an input failure.

Except in the case of a % specifier, the input item (or, in the case of a %n directive, the count of input wide characters) is converted to a type appropriate to the conversion specifier. If the input item is not a matching sequence, the execution of the directive fails: this condition is a matching failure. Unless assignment suppression was indicated by a *, the result of the conversion is placed in the object pointed to by the first argument following the format argument that has not already received a conversion result. If this object does not have an appropriate type, or if the result of the conversion cannot be represented in the space provided, the behaviour is undefined.

The following conversion specifiers are valid:

d	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *wcstol*( ) function with the value 10 for the base argument. The corresponding argument shall be a pointer to integer.
i	Matches an optionally signed integer, whose format is the same as expected for the subject sequence of the *wcstol*( ) function with the value 0 for the base argument. The corresponding argument shall be a pointer to integer.
0	Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the *wcstoul*( ) function with the value 8 for the base argument. The corresponding argument shall be a pointer to unsigned integer.
u	Matches an optionally signed decimal integer, whose format is the same as expected for the subject sequence of the *wcstoul*( ) function with the value 10 for the base argument. The corresponding argument shall be a pointer to unsigned integer.
x	Matches an optionally signed hexadecimal integer, whose format is the same as expected for the subject sequence of the *wcstoul*( ) function with the value 16 for the base argument. The corresponding argument shall be a pointer to unsigned integer.
e,f,g	Matches an optionally signed floating-point number, whose format is the same as expected for the subject sequence of the wcstod function. The corresponding argument shall be a pointer to floating.
s	Matches a sequence of non-white-space wide characters. If no l qualifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an the **mbstate_t** object pointed to by *ps*. The corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

Otherwise, the corresponding argument shall be a pointer to the initial element of an array of wchar_t type large enough to accept the sequence and a terminating null wide character, which will be added automatically.

[Matches a non-empty sequence of wide characters from a set of expected characters (the scanset). If no l qualifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence and a terminating null character, which will be added automatically.

If an l qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of wchar_t type large enough to accept the sequence and a terminating null wide character, which will be added automatically.

The conversion specifier includes all subsequent wide characters in the format string, up to and including the matching right bracket wide character ( ]). The wide characters between the brackets (the scanlist) comprise the scanset, unless the wide character after the left bracket is a circumflex (ˆ), in which case the scanset contains all wide characters that do not appear in the scanlist between the circumflex and the right bracket. If the conversion specifier begins with [ ] or [ˆ], the right bracket wide character is in the scanlist and the next right bracket wide character is the matching right bracket that ends the specification; otherwise the first right bracket wide character is the one that ends the specification. If a – wide character is in the scanlist and is not the first, nor the second where the first wide character is a ˆ, nor the last character, the behaviour is implementation-defined.

c Matches a sequence of wide characters of the number specified by the field width (1 if no field width is present in the directive). If no l qualifier is present, characters from the input field are converted as if by repeated calls to the *wcrtomb*( ) function, with the conversion state described by an **mbstate_t** object initialised to zero before the first wide character is converted. The corresponding argument shall be a pointer to a character array large enough to accept the sequence. No null character is added.

If an l qualifier is present, the corresponding argument shall be a pointer to the initial element of an array of wchar_t type large enough to accept the sequence. No null wide character is added.

p Matches an implementation-defined set of sequences, which should be the same as the set of sequences that may be produced by the %p conversion of the fwprintf function. The corresponding argument shall be a pointer to a pointer to void. The interpretation of the input item is implementation-defined. If the input item is a value converted earlier during the same program execution, the pointer that results shall compare equal to that value; otherwise the behaviour of the %p conversion is undefined.

n No input is consumed. The corresponding argument shall be a pointer to integer into which is to be written the number of wide characters read from the input stream so far by this call to the *m_swscanf*( ) function. Execution of a %n directive does not affect the assignment count returned at the completion of execution of the *m_swscanf*( ) function.

% Matches a single % ; no conversion or assignment occurs. The complete conversion specification shall be %%

If a conversion specification is invalid, the behaviour is undefined. The conversion specifiers E, G and X are also valid and behave the same as, respectively, e, g and x .

If end-of-file is encountered during input, conversion is terminated. If end-of-file occurs before any wide characters matching the current directive have been read (other than leading white space, where permitted), execution of the current directive terminates with an input failure; otherwise, unless execution of the current directive is terminated with a matching failure, execution of the following directive (other than %n, if any) is terminated with an input failure.

Trailing white space (including new-line wide characters) is left unread unless matched by a directive. The success of literal matches and suppressed assignments is not directly determinable other than via the %n directive.

RETURN VALUE

The *m_swscanf*( ) function returns the value of the macro EOF if an input failure occurs before any conversion. Otherwise, the *m_swscanf*( ) function returns the number of input items assigned, which can be fewer than provided for, or even zero, in the event of an early matching failure.

ERRORS

The *m_swscanf*( ) function may fail if:

[EBADF]
The **mbstate_t** object is invalid or was not created by a call to *m_creatembstate*( ).

APPLICATION USAGE

The *m_swscanf*( ) function enables an application to produce the same result as a call to *fwscanf*( ) on a stream which has been associated with an attribute object by a call to *m_fattr*( ).

SEE ALSO

m_sprintf( ), *m_sscanf*( ), *m_sscanf*( ).

CHANGE HISTORY

New for the DIS, with input from the MSE standard.

NAME

m_tombstrans — transliterate a character string using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wctype.h>

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

DESCRIPTION

The *m_tombstrans*( ) function maps the character string *inbuf* using the transliteration mapping described by *desc* into the array specified by *outbuf*. The setting of the LC_CTYPE category in the *attrobj* shall be the same as during the call to *m_wctrans*( ) or *wctrans*( ) that returned the value *desc*. Otherwise the result is implementation dependent.

The *m_tombstrans*( ) function is defined to operate on EOF and valid character encodings in the locale defined in *attrobj*. If any character code in *inbuf* is not in the domain of the locale defined in *attrobj*, the result is undefined. Any character code not included in the transliteration mapping described by *desc* are moved to the output buffer unchanged.

The *inbuf* argument points to a variable that points to the first character in the input buffer.

The *inbufleft* argument, on input, specifies the number of bytes to be mapped. A value of −1 indicates that the input is delimited by a NULL character.

The *outbuf* argument points to a variable that points to the first available character code in the output buffer.

The *outbufleft* argument is decremented to reflect the number of bytes still available in the output buffer. On return, the value is modified to reflect the number of wide-character codes left unfilled in the buffer. If the *outbufleft* argument is equal to zero, the *m_towcstrans*( ) function does not perform any transformation and returns the size of *outbufleft* needed to transform the contents of *inbuf*.

If the transliteration mapping encounters an invalid character code according to the locale defined in *attrobj*, mapping stops after the previous successfully mapped character code. If the input buffer ends with an incomplete composite sequence of character codes, conversion stops after the previous successfully mapped character code. If the output buffer is not large enough to hold the entire transliterated input, conversion stops just prior to the input character code that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the character code following the last character code used in the mapping. The value pointed to by *inbufleft* is decremented to reflect the number of bytes still not mapped in the input buffer. The variable pointed to by *outbuf* is updated to point to the character code following the last character code of mapped output data. The value pointed to by *outbufleft* is decremented to reflect the number of bytes still available in the output buffer. If *m_tombstrans*( ) encounters a character code in the input buffer that is legal, but for which a transliterated character code does not exist in the target mapping, *m_tombstrans*( ) copies the input character code into the output buffer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_tombstrans*( ) function updates the variables pointed to by the arguments to reflect the extent of the mapping and returns the number of non-transliterated mappings performed. If the entire string in the input buffer is mapped, the value pointed to by *inbufleft* is zero. If the transliteration mapping is stopped due to any condition mentioned above, the value pointed to by *inbufleft* is non-zero and *errno* is set to indicate the condition. If an error occurs *m_tombstrans*( ) returns (**size_t**)–1 and sets *errno*.

ERRORS

The *m_tombstrans*( ) function fails if:

[EILSEQ]
Transliteration mapping stopped due to an invalid character code that is not in the domain of the locale defined by *attrobj*.

[E2BIG]
Transliteration mapping stopped due to lack of space in the output buffer.

[EINVAL]
Transliteration mapping stopped due to an incomplete composite sequence of character codes at the end of the input buffer.

The *m_tombstrans*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *towcstrans*( ) function when called with the current locale set to the locale identified by *attrobj*, except that **char*** quantities are processed instead of wide characters.

The two strings — "upper" and "lower" — are reserved for the standard transliteration mappings.

The *m_tombstrans*( ) function provides the same set of transliteration mappings as are available from the *m_wctrans*( ) function for the same locale defined by *attrobj*. However, the reverse may not be true.

Note that the number of characters in *outbuf* may be different from *inbuf* when performing transliteration mapping on character strings containing composite sequences.

SEE ALSO

m_wctrans( ), *m_tombstrans*( ), *m_towcstrans*( ), *towlower*( ), *towctrans*( ), *towcstrans*( ), *towupper*( ), *wctrans*( ), **<mlocale.h>**, **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_towcstrans — transliterate a wide-character string using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>
#include <wctype.h>

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

DESCRIPTION

The *m_towcstrans*( ) function maps the wide-character string *inbuf* using the transliteration mapping described by *desc* into the array specified by *outbuf*. The setting of the LC_CTYPE category in the *attrobj* shall be the same as during the call to *m_wctrans*( ) or *wctrans*( ) that returned the value *desc*. Otherwise the result is implementation dependent.

The *m_towcstrans*( ) function is defined to operate on WEOF and wide-character codes corresponding to the valid character encodings in the locale defined in *attrobj*. If any wide-character code in *inbuf* is not in the domain of the locale defined in *attrobj*, the result is undefined. Any wide-character code not included in the transliteration mapping described by *desc* is moved to the output buffer unchanged.

The *inbuf* argument points to a variable that points to the first wide-character in the input buffer.

The *inbufleft* argument, on input, specifies the number of wide-character codes to be mapped. A value of −1 indicates that the input is delimited by a **wchar_t** NULL character.

The *outbuf* argument points to a variable that points to the first available wide-character code in the output buffer.

The *outbufleft* argument, on input, specifies the size of the output buffer (number of wide-character codes). On return, the value is modified to reflect the number of wide-character codes left unfilled in the buffer. If the *outbufleft* argument is equal to zero, the *m_towcstrans*( ) function does not perform any transformation and returns the size of *outbufleft* needed to transform the contents of *inbuf*.

If the transliteration mapping encounters an invalid wide-character code according to the locale defined in *attrobj*, mapping stops after the previous successfully mapped wide-character code. If the input buffer ends with an incomplete composite sequence of wide-character codes, conversion stops after the previous successfully mapped wide-character code. If the output buffer is not large enough to hold the entire transliterated input, conversion stops just prior to the input wide-character code that would cause the output buffer to overflow. The variable pointed to by *inbuf* is updated to point to the wide-character code following the last wide-character code used in the mapping. The value pointed to by *inbufleft* is decremented to reflect the number of wide-character codes still not mapped in the input buffer. The variable pointed to by *outbuf* is updated to point to the wide-character code following the last wide-character code of mapped output data. The value pointed to by *outbufleft* is decremented to reflect the number of wide-character codes still available in the output buffer. If *m_towcstrans*( ) encounters a wide-character code in the input buffer that is legal, but for which a transliterated wide-character code does not exist in the target mapping, *m_towcstrans*( ) copies the input wide-character code into the output buffer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_towcstrans*( ) function updates the variables pointed to by the arguments to reflect the extent of the mapping and returns the number of wide characters which were not modified by the transliteration mapping. If the entire string in the input buffer is mapped, the value pointed to by *inbufleft* is zero. If the transliteration mapping is stopped due to any condition mentioned above, the value pointed to by *inbufleft* is non-zero and *errno* is set to indicate the condition. If an error occurs *m_towcstrans*( ) returns (**size_t**)−1 and sets *errno*.

If the *outbufleft* argument is equal to zero, the *m_towcstrans*( ) function does not perform any transformation and returns the size of *outbufleft* needed to transform the contents of *inbuf*.

ERRORS

The *m_towcstrans*( ) function fails if:

[EILSEQ]
: Transliteration mapping stopped due to an invalid wide-character code that is not in the domain of the locale defined by *attrobj*.

[E2BIG]
: Transliteration mapping stopped due to lack of space in the output buffer.

[EINVAL]
: Transliteration mapping stopped due to an incomplete composite sequence of wide-character codes at the end of the input buffer.

The *m_towcstrans*( ) function may fail if:

[EBADF]
: The attribute object is invalid.

APPLICATION USAGE

The two strings — "upper" and "lower" — are reserved for the standard transliteration mappings.

The *m_towcstrans*( ) function provides the same set of transliteration mappings as are available from the *towcstrans*( ) function for the same locale defined by *attrobj*. However, the reverse may not be true.

Note that the number of characters in *outbuf* may be different from *inbuf* when performing transliteration mapping on wide-character strings containing composite sequences.

SEE ALSO

m_wctrans( ), *m_tombstrans*( ), *towlower*( ), *towctrans*( ), *towcstrans*( ), *towupper*( ), *wctrans*( ), **<mlocale.h>**. **<wchar.h>**, **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcscnt — count number of wide-character codes in a composite sequence using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcscnt*( ) function computes the number of wide-character codes in the composite sequence pointed to by *ptr*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If *ptr* is not a null pointer, the *m_wcscnt*( ) function returns 0 if *ptr* points to a null wide-character code or a combining character; otherwise it returns the number of wide-character codes including the non-combining wide-character code and zero or more following combining wide-character codes up to, but not including, the next non-combining wide-character code or terminating null wide-character code. If *ptr* is a null pointer, the *m_wcscnt*( ) function returns 0.

ERRORS

The *m_wcscnt*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

SEE ALSO

m_createattrobj( ), *m_wcsnext*( ), *m_wcsquery*( ), *m_wcswidth*( ), **<mlocale.h>**, **<wchar.h>**.

NAME

m_wcscoll — wide-character string comparison using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

int m_wcscoll(const AttrObject attrobj, const wchar_t *ws1,
              const wchar_t *ws2);
```

DESCRIPTION

The *m_wcscoll*( ) function compares the wide-character string pointed to by *ws1* to the wide-character string pointed to by *ws2*, both interpreted as appropriate to the LC_COLLATE category of the locale identified by *attrobj*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_wcscoll*( ) function returns an integer greater than, equal to or less than zero, according to whether the wide-character string pointed to by *ws1* is greater than, equal to or less than the wide-character string pointed to by *ws2* when both are interpreted as appropriate to the locale identified by *attrobj*. On error, *m_wcscoll*( ) may set *errno*, but no return value is reserved to indicate an error.

ERRORS

The *m_wcscoll*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The *ws1* or *ws2* arguments contain wide-character codes outside the domain of the collating sequence.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcscoll*( ) function when called with the current locale set to the locale identified by *attrobj*.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

The *m_wcsfxrm*( ) and *wcscmp*( ) functions should be used for sorting large lists.

SEE ALSO

wcscmp( ), *m_wcsxfrm*( ), *m_strcoll*( ), *m_strxfrm*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_wcscspn — get length of complementary wide substring

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

size_t m_wcscspn(const AttrObject attrobj, const wchar_t *ws1,
                 const wchar_t *ws2);
```

DESCRIPTION

The *m_wcscspn*( ) function computes the length of the maximum initial segment of the wide character string pointed to by *ws1* which consists entirely of wide-character codes *not* from the wide string pointed to by *ws2*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_wcscspn*( ) function returns the length of the complementary wide substring *ws1*; no return value is reserved to indicate an error.

ERRORS

The *m_wcscspn*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcscspn*( ) function when called with the current locale set to the locale defined by *attrobj*.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

SEE ALSO

m_wcsspn( ), *wcscspn*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcsfmon — convert monetary value to wide-character string using a locale object

SYNOPSIS

```
#include <mlocale.h>
#include <monetary.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcsfmon*( ) function places wide characters into the array pointed to by *ws* as controlled by the string pointed to by *format*. No more than *maxsize* wide characters are placed into the array.

The format is a character string that contains two types of objects: plain characters, which are converted to wide-characters and copied to the output stream; and conversion specifications, each of which results in the fetching of zero or more arguments which are converted and formatted. The results are undefined if there are insufficient arguments for the format. If the format is exhausted while arguments remain, the excess arguments are simply ignored.

This function behaves in the same manner as the *strfmon*( ) function when called with the current locale set to the locale identified by *attrobj*, except that a wide-character string is produced instead of a multi-byte string. Refer to the *strfmon*( ) function for a description of the conversion specification.

Locale Information

The LC_MONETARY category of the locale identified by *attrobj* affects the behaviour of this function including the monetary radix character (which may be different from the numeric radix character affected by the LC_NUMERIC category), the grouping separator, the currency symbols and formats. The international currency symbol should be conformant with the ISO 4217 standard.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If the total number of resulting wide characters including the terminating null wide-character code is not more than *maxsize*, the *m_wcsfmon*( ) function returns the number of wide-character codes placed into the array pointed to by *ws*, not including the terminating null wide-character code. Otherwise, −1 is returned, the contents of the array are indeterminate, and *errno* is set to indicate the error.

ERRORS

The *m_wcsfmon*( ) function fails if:

[E2BIG]
Conversion stopped due to lack of space in the buffer.

[EILSEQ]
Invalid byte sequence is detected during conversion to wide character string.

The *m_wcsfmon*( ) function may fail if:

[EBADF]
The attribute object is invalid.

EXAMPLES

Refer to the *strfmon*( ) for examples of different formats.

SEE ALSO

m_localeconv( ), *localeconv*( ), *strfmon*( ), *m_strfmon*( ), **<mlocale.h>**, **<monetary.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_wcsftime — convert data and time to wide-character string

SYNOPSIS

```
#include <mlocale.h>
#include <time.h>
#include <wchar.h>

size_t m_wcsftime(const AttrObject attrobj, wchar_t *wcs,
                  size_t maxsize, wchar_t *format,
                  const struct tm *timptr);
```

DESCRIPTION

The *m_wcsftime*( ) function places the wide-character codes into the array pointed to by *wcs* as controlled by the string pointed to by *format*.

This function behaves as if the character string generated by the *m_strftime*( ) function (when using the same locale identified by *attrobj*) is passed to the *mbsrtowcs*( ) function as the character string argument, and the *m_mbsrtowcs*( ) function places the result in the wide-character string argument of the *m_wcsftime*( ) function up to a limit of *maxsize* wide-character codes.

If copying takes place between objects that overlap, the behaviour is undefined.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If the total number of resulting wide-character codes including the terminating null wide-character code is no more than *maxsize*, the *m_wcsftime*( ) function returns the number of wide-character codes placed into the array pointed to by *wcs*, not including the terminating null wide-character code. Otherwise zero is returned and the contents of the array are indeterminate.

ERRORS

The *m_wcsftime*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcsftime*( ) function when called with the current locale set to the locale identified by *attrobj*. Refer to the *strftime*( ) function for a description of the conversion specification and application usage tips.

The format string is encoded in the file code of the locale identified by *attrobj*. Date and time formatting information is also extracted from this locale (category LC_TIME).

SEE ALSO

m_strftime( ), *m_strptime*( ), *m_setlocale*( ), *strftime*( ), *wcsftime*( ), **<mlocale.h>**, **<time.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification and Version 1 of this document.

NAME

m_wcsnext — advance to next composite sequence using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcsnext*( ) function locates the next non-combining wide-character code in a wide-character string.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If *ptr* is not a null pointer, the *m_wcsnext*( ) function either returns (wchar_t*)0 (if *ptr* points to a null wide-character code or a combining character), or returns a pointer to the next non-combining wide-character code in the string pointed to by *ptr* after skipping the composite sequence at the head of the string.

If *ptr* is a null pointer, the *m_wcsnext*( ) function returns the null pointer; that is (wchar_t*)0.

ERRORS

The *m_wcsnext*( ) function may fail if:

[EBADF]
    The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

SEE ALSO

m_createattrobj( ), **<mlocale.h>**, **<wchar.h>**.

NAME

m_wcspbrk — scan text object for wide-character code

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

wchar_t *m_wcspbrk(const AttrObject attrobj, const wchar_t *ws1,
                   const wchar_t *ws2);
```

DESCRIPTION

The *m_wcspbrk*( ) function locates the first occurrence in the wide-character string pointed to by *ws1* of any wide-character code from the wide-character string pointed to by *ws2*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

On successful completion, *m_wcspbrk*( ) returns a pointer to a wide-character code in *ws1* or a null pointer if no wide-character code from *ws2* occurs in *ws1*.

ERRORS

The *m_wcspbrk*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

SEE ALSO

m_setlocale( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcsptime — convert text to date and time object using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcsptime*( ) function converts the wide-character string pointed to by *ws* to values that are stored in the **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*. Date and time formatting information is also extracted from this locale (category LC_TIME).

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, *m_wcsptime*( ) returns a pointer to the wide-character string following the last wide-character code parsed. Otherwise, a null pointer is returned.

ERRORS

The *m_wcsptime*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *strptime*( ) function when called with the current locale set to the locale defined in *attrobj*. The *m_wcsptime*( ) function recognises the same conversion specifications as defined for the *strptime*( ) function (see the **XSH, Issue 4** specification).

SEE ALSO

m_setlocale( ), *m_strptime*( ), *m_wcsftime*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcsquery — query number of composite sequences using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcsquery*( ) function computes the number of composite sequences in the wide-character string pointed to by *ptr*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If *ptr* is not a null pointer, the *m_wcsquery*( ) function either returns 0 if *ptr* points to a null wide-character code or a combining character; otherwise it returns the number of composite sequences that consist of a non-combining wide-character code followed by zero or more combining wide-character codes up to, but not including, the terminating null wide-character code.

If *ptr* is a null pointer the *m_wcsquery*( ) function returns 0.

ERRORS

The *m_wcsquery*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

SEE ALSO

m_createattrobj( ), **<mlocale.h>**, **<wchar.h>**.

NAME

m_wcsscanfor — scan a wide-character string for a wide-character code

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcsscanfor*( ) function scans the wide-character string pointed to by *ws* at the offset specified by *position* for the first wide-character code that matches or does not match the set of classification criteria specified by *condition*.

The argument *condition* specifies a set of classification criteria that can be ORed bitwise. The following conditions are reserved for the standard classification criteria and are defined below in the **APPLICATION USAGE** section.

```
typedef enum { Alphabetic, WhiteSpace, Control, Digit, Graphic,
              Lowercase, Uppercase, Printing, Punctuation, HexDigit,
              LineBreakCharacter, LineBreakHyphen, LineBreakScript,
              WordBoundary, SentenceBoundary, ParagraphBoundary,
              CharsetBoundary, ScriptBoundary, CompositeBoundary
             } ScanCondition;
```

The argument *inverse* determines the rules for scanning, and is an integer type containing one of the values defined for the following enumeration type:

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

If *inverse* is set to False, the function searches for the first wide-character code that matches the specified condition; if *inverse* is set to True, the function searches for the first wide-character code that does not match the condition.

The argument *direction* determines the direction in which scanning takes place. It is an integer type containing one of the values defined for the following enumeration type:

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

If *direction* is set to Forw, the function scans from *position* to the end of the wide-character string *ws*; if *direction* is set to Back, the function scans from *position* to the beginning of the wide-character string *ws*.

The rules of any classification criteria are determined by the locale defined by *attrobj*.

The search begins from the wide-character code identified by *ws*, and continues through the next *num_chars* wide-character code.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If all the criteria specified by *condition* are met, a non-negative integer identifying the location of the first wide-character code meeting the criteria is returned. The non-negative integer value returned indicates the offset number of wide-character codes from the wide-character code identified by *ws*, to the wide-character code at which the condition is satisfied. If no wide character meets the specified criteria, a (**size_t**)−1 is returned and *errno* is set to indicate the error.

ERRORS

The *m_wcsscanfor*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The *condition* is invalid.

[EILSEQ]
No *condition* could be found in the supplied wide-character string.

APPLICATION USAGE

It is recommended that the *position* value be updated on subsequent calls since certain languages (for example, Thai) may need to view the code elements that precede *position* (when direction = Forw).

The set of classification criteria to be provided are defined as:

Alphabetic	same as isalpha
WhiteSpace	same as isspace
Control	same as iscntrl
Digit	same as isdigit
Graphic	same as isgraph
Lowercase	same as islower
Uppercase	same as isupper
Printing	same as isprint
Punctuation	same as ispunct
HexDigit	same as isxdigit
LineBreakCharacter	a wide-character code that is defined to cause a line discontinuity.
LineBreakHyphen	the next wide-character code after which a line discontinuity may occur due to hyphenation
LineBreakScript	the next wide-character code after which a line discontinuity may occur due to script definitions
WordBoundary	the next wide-character code after which the current language unit (word)
SentenceBoundary	the next wide-character code after which the current sentence
ParagraphBoundary	the next wide-character code after which the current paragraph
CharsetBoundary	the next wide-character code after which the current charset
ScriptBoundary	the next wide-character code after which the current script
CompositeBoundary	the next wide-character code after which the current composite sequence.

The *WordBoundary* identifies a *language unit* that is defined to be composed of one or more morphemes with relative freedom to enter into syntactic constructions by a particular implementation. The *WordBoundary* may be used by applications to determine sequences of code elements that should be treated as a single unit and may not be broken up. In effect, a language unit is either the smallest unit susceptible to independent use, or it consists of two or three such units combined under certain linking conditions that are implementation dependent. The language unit may be broken down into smaller elements by using the *LineHyphenBoundary* condition. For example, the *WordBoundary* takes into account that *black bird* is different from *black-bird*.

The conditions starting with *Line* are defined for identifying boundaries where lines may be discontinued or broken when presented. The *LineBreakCharacter* condition is defined for specific characters defined as line breaks, for example, the newline character. The *LineHyphenBoundary* condition is defined for cases where a line discontinuity may occur but specifically where a hyphen may be used or exist if a line discontinuity is performed. The *LineBreakBoundary* condition is defined for linguistic cases that define *line breakable* conditions not defined by any other condition. An example of a line breakable condition not covered by any of the other conditions is that Japanese phrases consisting of Kinsoku characters may or may not appear at the end of line or at the beginning of line. Such characters help to define boundary conditions that are treated as a line discontinuity.

The *CharsetBoundary* and *ScriptBoundary* are intended for advance text processing across multiple languages and writing systems. A charset boundary is used to segment [what?] based the locale's set of charset as defined by X11. The script boundary is to introduce the notion of multiple languages and writing systems within a string of code elements. These two conditions are viewed as different since the Japanese script can be considered to be made up of multiple charsets (JIS 0201, JIS 0208 and user-defined characters).

The boundary conditions *WordBoundary*, *SentenceBoundary*, *CharsetBoundary*, *ScriptBoundary*, *ParagraphBoundary* and *CompositeBoundary* use the wide-character pointed to by *ws* as the first character of the *current condition*. The *boundary* is the wide character found that does or does not match the current condition. In the case of *ScriptBoundary* and *CharsetBoundary*, where there may be multiple sets per locale, it is left to other layers to determine the specific member using the characters within the segment returned from the *m_wcsscanfor*( ) function.

SEE ALSO

m_strscanfor( ), *m_setlocale*( ), *m_iswctype*( ), **<mlocale.h>**, **<wchar.h>**.

NAME

m_wcsspn — get length of wide substring using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

size_t m_wcsspn(const AttrObject attrobj, const wchar_t *ws1,
                const wchar_t *ws2);
```

DESCRIPTION

The *m_wcsspn*( ) function computes the length of the maximum initial segment of the wide character string pointed to by *ws1* which consists entirely of wide-character codes from the wide string pointed to by *ws2*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_wcsspn*( ) function returns the length of a wide substring, *ws1*, using a locale object; no return value is reserved to indicate an error.

ERRORS

The *m_wcsspn*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

Because no return value is reserved to indicate an error, an application wishing to check for error situations should set *errno* to 0, then call *m_strcspn*( ), then check *errno* and if it is non-zero, assume an error has occurred.

This function behaves in the same manner as the *wcsspn*( ) function when called with the current locale set to the locale defined by *attrobj*. The locale defined in *attrobj* may be used in matching characters in cases where different sequences of combining character may be used to define a character.

SEE ALSO

wcscspn( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcstod — convert text to double-precision number using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

double m_wcstod(const AttrObject attrobj, const wchar_t *src,
                wchar_t **end_ptr);
```

DESCRIPTION

The *m_wcstod*( ) function converts the initial portion of the wide-character string pointed to by *src* to double representation. First, it decomposes the input into three parts: an initial (possibly empty) sequence of white-space wide-character codes (as specified by the *iswspace*( ) function), a subject sequence resembling a floating-point constant, and a final sequence of one or more unrecognised wide-character codes, including the terminating null wide-character code. Then, it attempts to convert the subject sequence to a floating point-number, and returns the result.

The expected form of the subject sequence is an optional plus or minus sign, then a non-empty sequence of digits optionally containing a decimal-point, then an optional exponent part as defined for the corresponding single-byte characters in subclause 6.1.3.1 of the ISO C standard, but no floating suffix. The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code that is not white space and that is of the expected form. The subject subsequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white space, or if the first wide-character code that is not white space is other than a sign, a digit or a decimal-point.

If the subject sequence has the expected form, the sequence of wide-character codes starting with the first digit or the decimal-point wide-character code (whichever occurs first) is interpreted as a floating constant according to the rules of subclause 6.1.3.1 of the ISO C standard, except that the decimal-point wide-character code is used in place of a period, and that if neither an exponent part nor a decimal-point wide-character appears, a decimal point is assumed to follow the last digit in the wide-character string. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final sequence is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Additional implementation-dependent subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *src* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_wcstod*( ) function returns the converted value, if any. If no conversion could be performed, zero is returned, and *errno* may be set to [EINVAL]. If the correct value is outside the range of representable values, ±HUGE_VAL is returned (according to the sign of the value) and [ERANGE] is returned as a status to indicate an out of range condition. If the correct value would cause underflow, zero is returned and *errno* is set to [ERANGE].

ERRORS

The *m_wcstod*( ) function fails if:

[ERANGE]
    The value to be returned would cause overflow or underflow.

The *m_wcstod*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
No conversion could be performed.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcstod*( ) function when called with the current locale set to the locale defined by *attrobj*.

Number formatting information is extracted from the locale associated with *attrobj*(category LC_NUMERIC). In other than the POSIX locale, additional implementation-defined subject sequence forms may be accepted.

SEE ALSO

m_createattrobj( ), *m_setlocale*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcstok — split wide-character string into tokens using a text context object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

wchar_t *m_wcstok(wchar_t *ws1, const wchar_t *ws2, mbstate_t* ps)
```

DESCRIPTION

A sequence of calls to the *m_wcstok*( ) function breaks the wide-character string pointed to by *ws1* into a sequence of tokens, each of which is delimited by a wide-character code from the wide-character string pointed to by *ws2*. The *ps* argument points to a text context object of type **mbstate_t** that is used to store any tokenising context necessary for *m_wcstok*( ) to continue scanning the same wide-character string.

For the first call in the sequence, *ws1* shall point to a wide string, while in subsequent calls for the same string, *ws1* shall be a null pointer. If *ws1* is a null pointer, the value pointed to by *ps* shall match that stored by the previous call for the same wide-character string. The separator wide-character string pointed to by *ws2* may be different from call to call.

The first call in the sequence searches the wide-character string pointed to by *ws1* for the first wide-character code that is not contained in the current separator string pointed to by *ws2*. If no such wide-character code is found, then there are no tokens in the wide-character string pointed to by *ws1* and the *m_wcstok*( ) function returns a null pointer. If such a wide-character code is found, it is the start of the first token.

The *m_wcstok*( ) function then searches from there for a wide-character code that *is* contained in the current separator wide-character string. If no such wide-character code is found, the current token extends to the end of the wide-character string pointed to by *ws1*, and subsequently searches in the same wide-character string for a token and returns a null pointer. If such a wide-character code is found, it is overwritten by a null wide-character code, which terminates the current token.

In all cases, the *wcstok*( ) function stores sufficient information in the type **mbstate_t** pointed to by *ps* so that subsequent calls, with a null pointer for *ws1*, start searching just past the element overwritten by a null wide-character code (if any).

RETURN VALUE

Upon successful completion, the *m_wcstok*( ) function returns a pointer to the first wide-character code of a token. Otherwise, if there is no token, *m_wcstok*( ) returns a null pointer.

ERRORS

No errors are defined.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcstok*( ) function when called with the current locale set to the locale defined by *attrobj*.

The encoding of wide-character codes is defined by the locale associated with *ps*. An example of searching for tokens in a multi-locale environment is as follows:

```
#include <mlocale.h>

static wchar_t str1[] = L"?a???b,,,#c";
static wchar_t str2[] = L"\t \t";
wchar_t *t;
AttrObject foo_locale;            /* previously initialised */
AttrObject C_locale;              /* previously initialised */
mbstate_t ps1 = m_creatembstate(foo_locale );
mbstate_t ps2 = m_creatembstate(C_locale );

t = m_wcstok(str1, L"?", &ps1);    /* t points to the token L"a" */
t = m_wcstok(NULL, L",", &ps1);    /* t points to the token L"??b" */
t = m_wcstok(str2, L"\t", &ps2);  /* t points to L */
t = m_wcstok(NULL, L"#", &ps1);    /* t points to the token L",," */
t = m_wcstok(NULL, L"?", &ps2);    /* t is a null pointer */
t = m_wcstok(NULL, L"?", &ps1);    /* t is a null pointer */

m_destroymbstate(ps1 );
m_destroymbstate(ps2 );
```

SEE ALSO

m_creatembstate( ), *m_setlocale*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcstol — convert text to long integer using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

long m_wcstol(const AttrObject attrobj, const wchar_t *src,
              wchar_t **end_ptr, int base)
```

DESCRIPTION

The *m_wcstol*( ) function converts the initial portion of the wide-character string pointed to by *src* to type **long int** representation. First, it decomposes the input into three parts: an initial (possibly empty) sequence of white-space wide-character codes (as specified by the *iswspace*( ) function), a subject sequence resembling an integer represented in some radix determined by the value of *base*( ), and a final sequence of one or more unrecognised wide-character codes, including the terminating null wide-character code of the input wide-character string. Then, it attempts to convert the subject sequence to an integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in the ISO C standard optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) to z (or Z) are ascribed the values 10 to 35; only letters and digits whose ascribed values are less than that of *base* are permitted. If the value of base is 16, the wide-character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input string, starting with the first wide-character code that is not white space and that is of the expected form. The subject sequence contains no data if the input wide-character string is empty or consists entirely of white space, or if the first wide-character code that is not white space is other than a sign, or permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant according to the rules of the ISO C standard. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final sequence is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *src* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_wcstol*( ) function returns the converted value, if any. If no conversion could be performed, zero is returned, and *errno* may be set to [EINVAL].

If the correct value is outside the range of representable values, {LONG_MAX} or {LONG_MIN} is returned (according to the sign of the value) and *errno* is set to [ERANGE] to indicate an out of range condition.

ERRORS

The *m_wcstol*( ) function fails if:

[ERANGE]
The value to be returned is not representable.

The *m_wcstol*( ) function may fail if:

[EINVAL]
The value of *base* is not supported.

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcstol*( ) function when called with the current locale set to the locale defined by *attrobj*.

Number formatting information is extracted from the locale associated with *attrobj* (category LC_NUMERIC). In other than the POSIX locale, additional implementation-defined subject sequence forms may be accepted.

SEE ALSO

m_createattrobj( ), *m_setlocale*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcstoul — convert text to unsigned long integer using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

unsigned long m_wcstoul(const AttrObject attrobj, const wchar_t *src,
                        wchar_t *endptr, int base);
```

DESCRIPTION

The *m_wcstoul*( ) function converts the initial portion of the wide-character string pointed to by *src* to type **unsigned long int** representation. First, it decomposes the input into three parts: an initial (possibly empty) sequence of white-space wide-character codes (as specified by the *iswspace*( ) function), a subject sequence resembling an integer represented in some radix determined by the value of *base*( ), and a final sequence of one or more unrecognised wide-character codes, including the terminating null wide-character code of the input wide-character string. Then, it attempts to convert the subject sequence to an unsigned long integer, and returns the result.

If the value of *base* is zero, the expected form of the subject sequence is that of an integer constant as described for the corresponding single-byte characters in subclause 6.1.3.2 of the ISO C standard optionally preceded by a plus or minus sign, but not including an integer suffix. If the value of *base* is between 2 and 36 (inclusive), the expected form of the subject sequence is a sequence of letters and digits representing an integer with the radix specified by *base*, optionally preceded by a plus or minus sign, but not including an integer suffix. The letters from a (or A) to z (or Z) are ascribed the values 10 to 35; only letters whose ascribed values are less than that of *base* are permitted. If the value of *base* is 16, the wide-character codes 0x or 0X may optionally precede the sequence of letters and digits, following the sign if present.

The subject sequence is defined as the longest initial subsequence of the input wide-character string, starting with the first wide-character code character that is not white space, that is of the expected form. The subject sequence contains no wide-character codes if the input wide-character string is empty or consists entirely of white space, or if the first wide-character code that is not white space is other than a sign, or permissible letter or digit.

If the subject sequence has the expected form and the value of *base* is zero, the sequence of wide-character codes starting with the first digit is interpreted as an integer constant according to the rules of subclause 6.1.3.2 of the ISO C standard. If the subject sequence has the expected form and the value of *base* is between 2 and 36, it is used as the base for conversion, ascribing to each letter its value as given above. If the subject sequence begins with a minus sign, the value resulting from the conversion is negated. A pointer to the final sequence is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

Additional implementation-defined subject sequences may be accepted.

If the subject sequence is empty or does not have the expected form, no conversion is performed; the value of *src* is stored in the object pointed to by *endptr*, provided that *endptr* is not a null pointer.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

The *m_wcstoul*( ) function returns the converted value, if any. If no conversion could be performed, zero is returned and *errno* may be set to [EINVAL]. If the correct value is outside the range of representable values, {ULONG_MAX} is returned and *errno* is set to [ERANGE].

ERRORS

The *m_wcstoul*( ) function fails if:

[ERANGE]
The value to be returned is not representable.

The *m_wcstoul*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The value of *base* is not supported.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcstoul*( ) function when called with the current locale set to the locale defined by *attrobj*.

Number formatting information is extracted from the locale associated with *attrobj* category LC_NUMERIC). In other than the POSIX locale, additional implementation-defined subject sequence forms may be accepted.

SEE ALSO

m_setlocale( ), *m_wcstod*( ), *m_wcstol*( ), *m_wcstoul*( ), *wcstod*( ), *wcstol*( ), *wcstoul*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcswcs — find wide substring

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

wchar_t *m_wcswcs(const AttrObject attrobj, const wchar_t *ws1,
                  const wchar_t *ws2);
```

DESCRIPTION

The *m_wcswcs*( ) function locates the first occurrence in the wide character string pointed to by *ws1* of the sequence of wide-character codes (excluding the terminating null wide-character code) in the wide character string pointed to by *ws2*.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion, the *m_wcswcs*( ) function returns a pointer to the located wide character string or a null pointer if the wide-character string is not found.

If *ws2* points to a wide character string with zero length, the function returns *ws1*.

ERRORS

The *m_wcswcs*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcswcs*( ) function when called with the current locale set to the locale defined by *attrobj*. The locale defined in *attrobj* may be used to match characters in cases where different sequences of combining characters may be used to define a character.

SEE ALSO

wcschr( ), *wcswcs*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wcswidth — query width of a wide-character string using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

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

DESCRIPTION

The *m_wcswidth*( ) function determines the number of presentation column positions required for *n* wide-character strings (or fewer wide-character strings if a null wide-character code is encountered before the n wide-character strings are exhausted) in the string pointed to by *ptr*.

Note: This function assumes that the 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.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If *ptr* is not a null pointer, the *m_wcswidth*( ) function either returns 0 if *ptr* points to a null wide-character code or a combining character; otherwise it returns the number of display columns to be occupied by the *n* or fewer wide-character strings of the string pointed to by *ptr*.

If *ptr* is a null pointer the *m_wcswidth*( ) function returns 0.

ERRORS

The *m_wcswidth*( ) function may fail if:

[EBADF]
The attribute object is invalid.

SEE ALSO

m_createattrobj( ), **<mlocale.h>**, **<wchar.h>**.

NAME

m_wcsxfrm — associate collating weights with wide-character string using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wchar.h>

size_t m_wcsxfrm(const AttrObject attrobj, wchar_t *ptr1,
                 const wchar_t *ptr2, size_t n);
```

DESCRIPTION

The *m_wcsxfrm*( ) function transforms the wide-character string pointed to by *ptr2* and places the resulting transformed wide string into the array pointed to by *ptr1*. The transformation is such that if the *wcsncmp*( ) function is applied to two transformed wide strings, it returns a value greater than, equal to or less than zero, corresponding to the result of the *m_wcscoll*( ) function applied to the same two original wide-character strings. If *n* is zero, *ptr1* is permitted to be a null pointer.

No more than *n* elements are placed into the resulting array pointed to by *ptr1*, including the terminating null wide-character code. If copying takes place between objects that overlap the behaviour is undefined.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

Upon successful completion the *m_wcsxfrm*( ) function returns the length of the transformed string (not including the terminating null wide-character code). If the value returned is equal to or more than the input value of *n*, the contents of the wide-character codes pointed to by *ptr1* are indeterminate.

If *ptr1* is a null pointer, the *m_wcsxfrm*( ) function returns the number of elements required to contain the transformed character string.

On error, the *m_wcsxfrm*( ) function returns (**size_t**)–1**,** and sets *errno* to indicate the error.

ERRORS

The *m_wcsxfrm*( ) function may fail if:

[EBADF]
The attribute object is invalid.

[EINVAL]
The wide-character string pointed to by *ptr2* contains wide-character codes outside the domain of the collating sequence.

APPLICATION USAGE

Note: The value of a wide character FOO created in locale A may be different from wide character FOO created in locale B. Therefore the locale of operation must match that used to create the wide character; otherwise the results of the operation are undefined.

This function behaves in the same manner as the *wcsxfrm*( ) function when called with the current locale set to the locale defined by *attrobj*.

SEE ALSO

m_wcscoll( ), *m_setlocale*( ), *wcscoll*( ), *wcsxfrm*( ), *wcsncmp*( ), **<mlocale.h>**, **<wchar.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wctrans — define character transliteration using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wctype.h>

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

DESCRIPTION

The *m_wctrans*( ) function constructs a value with the type **wctrans_t** that describes a transliteration of wide characters identified by the string argument *property*.

The setting of the LC_CTYPE category in the *attrobj* shall be the same as during subsequent calls to *m_tombstrans*( ) or *m_towcstrans*. Otherwise the result is implementation dependent.

The two strings listed in the description of the *m_towcstrans*( ) and *m_tombstrans*( ) functions shall be valid in all locales as the *property* argument to the *m_wctrans*( ) function.

Additional character transliteration names defined in the locale definition file (category LC_CTYPE) can also be specified.

If *attrobj* is defined as (**AttrObject**)NULL, the behaviour of the function is defined by the current (global) locale setting as defined by the *setlocale*( )function.

RETURN VALUE

If *property* identifies a valid transliteration of wide-character and multi-byte character codes according to the LC_CTYPE category of the locale identified by *attrobj*, the *m_wctrans*( ) function returns a non-zero value that is valid as the *desc* argument to subsequent calls of *m_tombstrans*( ) or *m_towcstrans*( ); otherwise, it returns zero.

ERRORS

The *m_wctrans*( ) function may fail if:

[EBADF]
The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *wctrans*( ) function when called with the current locale set to the locale defined by *attrobj*.

The *property* argument is a string identifying a generic transliteration for which codeset-specific information is required.

SEE ALSO

m_tombstrans( ), *m_towcstrans*( ), *wctype*( ), *isctype*( ), *towupper*( ), *towlower*( ), **<mlocale.h>**, **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

NAME

m_wctype — define character class using locale object

SYNOPSIS

```
#include <mlocale.h>
#include <wctype.h>

wctype_t m_wctype(const AttrObject attrobj, const char *property);
```

DESCRIPTION

The *m_wctype*( ) function constructs a value with the type **wctype_t** that describes a class of characters identified by the string argument *property*.

The eleven strings listed in the description of the *m_iswctype*( ) function shall be valid in all locales as *property* argument to the *m_wctype*( ) function.

Additional character class names defined in the locale definition file (category LC_CTYPE) can also be specified.

RETURN VALUE

If *property* identifies a valid class of character according to the LC_CTYPE category of the locale identified by *attrobj*, the *m_wctype*( ) function returns a non-zero value that is valid as the argument of type **wctype_t** to subsequent calls of *m_iswctype*( ), *m_isctype*( ), *iswctype*( ); otherwise, it returns zero.

ERRORS

The *m_wctype*( ) function may fail if:

[EBADF]
    The attribute object is invalid.

APPLICATION USAGE

This function behaves in the same manner as the *wctype*( ) function when called with the current locale set to the locale defined by *attrobj*.

The setting of the LC_CTYPE category in the *attrobj* shall be the same as during subsequent calls to *m_iswctype*( ), *m_isctype*( ) or *iswctype*( )). Otherwise the result is implementation dependent.

The *property* argument is a string identifying a generic character class for which codeset-specific information is required.

SEE ALSO

m_iswctype( ), *m_isctype*( ), *wctype*( ), *isctype*( ), *iswalnum*( ), *iswalpha*( ), *iswcntrl*( ), *iswdigit*( ), *iswgraph*( ), *iswlower*( ), *iswprint*( ), *iswpunct*( ), *iswspace*( ), *iswupper*( ), *iswxdigit*( ), **<mlocale.h>**, **<wctype.h>**.

CHANGE HISTORY

Derived from the **XSH, Issue 4** specification, Version 1 of this document and the MSE standard.

Appendix A

Locale Registry

X/Open's Locale Registry contains locales that have been accepted as conforming to the rules in the **Locale Registry Procedures** guide. These locales are available from X/Open in electronic form. An index of available locales is published in the **Publications Price List**.

The registration of private locales is permitted where a national standard locale is not available. Any such registration is subject to withdrawal of the locale when a national standard becomes available.

Values for the data type **LocaleNetToken** are defined in the X/Open Locale Registry.

Values for the data type **LocaleNetString** shall conform to the string network locale specification syntax as described on Section 3.4 on page 16.

The corresponding convention shall be used for portions of a **LocaleNetString** whose locales has been registered under the X/Open Locale Registry:

```
register_spec  := 'XOPEN'
name_spec      := name of locale in X/Open Registry
encoding_spec  := XFN_encoding that corresponds to the
                  Federated Naming registry for encodings.
```

A few examples of values for **HostLocaleString** and it's corresponding **LocaleNetToken** are shown in the following table. The **LocaleNetString** for the German locale can be found in Example 3-4 on page 19.

Locale	**HostLocaleString**	**LocaleNetToken**
German for Germany	de_DE	003
Icelandic for Iceland	is_IS	006
Japanese for Japan	ja_JP	001

Appendix B

Alternatives Examined and Rationale

This appendix describes other proposals considered for locale management, along with the rationale for their rejection.

B.1 Locale Object per Category

It was proposed that a handle for each category in a locale object be visible to the application. For example, the application might create an LC_CTYPE object in locale **foo**, an LC_COLLATE object in locale **bar**, and the remaining locale objects in locale **woof**. The reasoning for this was that some applications only need the capabilities represented by one or two categories; this provided a way to have minimum data size impact on the application, as well as reducing process time to initialise the locale object. For example, an application that only needs to parse string data might only need to call a function like *mblen*( ), and thus only needs the LC_CTYPE portion of the locale. The specification included a procedure like *mblen*( ) that took as its argument a pointer to an LC_CTYPE locale object.

This proposal was rejected because of the number of new APIs it introduced. The proposal necessitated an API for each locale category object to be created, one API for each locale category object to be destroyed, plus one new API for each existing locale-sensitive function in today's architecture. Given that there are currently seven LC_* categories, with the possibility for that number to grow, this would force the addition of at least 14 new APIs and at least two APIs every time a new category is added.

B.2 Locale Object Method

Another proposal required that the locale object be specified to contain at least one public method; the method accepted a locale object handle, a token identifying which function was to be performed, and a *varargs* argument with all the data necessary to perform the operation. The proposal also used opaque data types in the *varargs* argument list, allowing for future expansion to tagged data without an API change. The reasoning for the proposal was that it produced only one new API for each internationalisation function, and did not require new APIs for future capability.

The proposal was rejected because it was extremely C-language specific and because it would force a severe change in current programming style (current internationalisation functions return their results directly; this proposal would force them to be returned in an argument).

B.3 No Change

The group discussed at length making no change at all. It was argued successfully that with the changes necessary to handle stateful encodings, and with the proposed changes to *setlocale*( ) for threads in POSIX.1c, it would be possible to implement multi-lingual or multi-locale applications without any changes or additions to the current architecture.

The group decided that this was not acceptable. The current specifications had been loose enough to permit vendors to implement *setlocale*( ) as a heavy-weight procedure. The group felt that application developers needed a way to ensure light-weight operation. The group also felt that forcing the application to manage global state was encouraging poor program design and that it was incompatible with the object-oriented paradigm. Finally, the group was convinced that publicly available locale handles will be necessary for handling tagged data.

B.4 Locale Objects for Threads

The group had originally proposed locale objects as a solution for the problem with a global locale in threaded applications. However, members of the group felt that the thread problem could be solved with a simple semantic alteration to the definition of *setlocale*( ) in the POSIX.1c specification.

B.5 Linking Global Locale with Non-global Locale

In an effort to gain leverage from the existing internationalisation functions, the group considered at length a proposal that allowed the current global locale to be swapped out and a non-global locale object to be swapped in as the global locale. The functions that performed these services were: *locale_get_current*( ) and *locale_switch*( ). The former function looked at the current global locale, created a locale object identical to the current global locale, and returned a pointer to that new object; the latter function took a locale object and made it global (that is, so that existing system interfaces would use its data to perform their operations). Combined, these functions gave an application a guaranteed light-weight way to swap locales. An additional benefit of this approach was that it allowed the locale management functions to be used independently of any functions that might take a locale object as argument, or that might use a locale object in a data tag.

In rejecting this approach, the group stated that the non-global locale model really was a new model and that there was no need to link the two models. Additionally, in working on the details of which categories would be affected, the idea of a *simple swap* became not so simple. For example, suppose that the application's global locale had been set with:

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

using the environment variables to determine the settings of each of the LC_ categories. Now further suppose that the application creates an incomplete locale object (that is, LC_CTYPE and LC_COLLATE are specified, but other categories are allowed to default). The group ended up with a very complicated model when they tried to define the semantics of *locale_switch*( ) when applied to this environment; at least, there was no definition that was intuitively obvious to everyone in the group. No-one in the group relished trying to describe the effects to those who are not internationalisation experts.

The group decided that there was little gain in tying the two models together. Internationalisation is a baffling technology for most people; if a group of internationalisation experts had trouble trying to visualise and work with mixed models, it was felt that there was little value for the typical application writer. Further, the group felt that there was significant cost in trying to mix the models; backwards compatibility would constrain the ability to utilise fully capabilities offered by the new model. The group felt it best simply to offer the two models as different programming paradigms, emphasising the advantages and costs of each.

B.6 Opaque Data Functions

The solution proposed in Version 1 of this document is a set of functions that provide similar capabilities to those provided in the existing ISO C and POSIX standards, and the **XSH, Issue 4** specification. The naming convention adopted is to prefix new interfaces with *o_*. Interfaces have also been introduced in other areas, either to provide management functions for new objects, or to replace standard interfaces with functions specific to the multi-locale model. In all cases, the *o_* prefix is used to indicate that the proposed functions are capable of dealing with opaque text objects and multiple locales.

Also added to the arguments passed to the global locale functions is either a locale object handle of type **LocaleObject**, or an attribute object handle of type **AttrObject**. Character or wide-character types in the **XSH, Issue 4** specification are replaced in the multi-locale model by pointers to text objects of type **txt_ptr**.

B.6.1 Objectives

The objectives of the multi-locale support functions are to:

- satisfy multi-lingual, multi-threading, multi-node processing
- satisfy the multi-locale support requirements encountered in the windowing environment
- co-exist with the global locale functions
- address the problem of stateful encodings, context-sensitive rendering and multi-directional text
- address the limitations that reliance on global data places on object-oriented programming paradigms.

B.6.2 Assumptions

The multi-locale support functions are based on the assumption that the Locale Registry exists.

B.6.3 Reasons for Rejection

The opaque data concept proved unpopular. Implementors could see no demand for systems using opaque data; application developers disliked the idea of not being able to access the data directly. Many felt that the concept addressed the needs of a small percentage of the market, but was too complex for the majority.

Glossary

byte

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

character

A sequence of one or more bytes representing a single graphic symbol or control code. This term corresponds to the ISO C standard term *multibyte character* (multi-byte character), where a single-byte character is a special case of a multi-byte character. Unlike the usage in the ISO C standard, *character* here has no necessary relationship with storage space, and *byte* is used when storage space is discussed.

character class

A named set of characters sharing an attribute associated with the name of the class. The classes and the characters that they contain are dependent on the value of the LC_CTYPE category in the current locale.

character set

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

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 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.

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*.

collating element

The smallest entity used to determine the logical ordering of character or wide character strings. See **collation sequence** on page 142. A collating element consists of either a single character, or two or more characters collating as a single entity. The value of the LC_COLLATE category in the current locale determines the current set of collating elements.

collation
The logical ordering of character or wide-character strings according to defined precedence rules. These rules identify a collation sequence between the collating elements, and such additional rules that can be used to order strings consisting of multiple collating elements.

collation sequence
The relative order of collating elements as determined by the setting of the LC_COLLATE category in the current locale. The character order, as defined for the LC_COLLATE category in the current locale, defines the relative order of all collating elements, such that each element occupies a unique position in the order. This is the order used in ranges of characters and collating elements in regular expressions and pattern matching. In addition, the definition of the collating weights of characters and collating elements uses collating elements to represent their respective positions within the collation sequence.

Multi-level sorting is accomplished by assigning elements one or more collation weights, up to the limit {COLL_WEIGHTS_MAX}; see **<limits.h>** in the **XSH, Issue 4** specification. On each level, elements may be given the same weight (at the primary level, called an equivalence class; see **equivalence class**) or be omitted from the sequence. Strings that collate equal using the first assigned weight (primary ordering) are then compared using the next assigned weight (secondary ordering), and so on.

composite sequence
One or more code elements that together form 1 graphic or control code.

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

current locale
See **global locale** on page 143.

current position index
The current position index indicates the absolute position of a text character within a text object. Index 0 (zero) identifies the first text character in such an object.

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

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

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

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

file code
The representation of text when it is stored on some external storage medium (for example, magnetic disk). File codes are implementation-defined. Functions exist to convert between file codes and process codes either implicitly (text stream I/O) or explicitly (*o_mbstowcs*( ), and so on).

global locale
The particular locale on a host system whose contents (information, data or processing) are visible to an entire process on a single host system.

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

host locale string
A character string that represents a given locale on a particular implementation. A host locale string may be:

- an alias for a locale database
- the name for a locale database
- the string returned by:

```
setlocale(LC_ALL,NULL)
```

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.

local customs
The conventions of a geographical area or territory for such things as date, time and currency formats.

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

locale object
The abstraction for representing the contents of a particular locale that is known as a host system object. On a host system, a locale object is of type **AttrObject**.

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

message catalogue
A file or storage area containing program messages, command prompts and responses to prompts for a particular native language, territory and codeset.

native language
A computer user's spoken or written language, such as American English, British English, Danish, Dutch, French, German, Italian, Japanese, Norwegian or Swedish.

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 142.

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.

radix character

The character that separates the integer part of a number from the fractional part.

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.

text context object

An abstraction for representing stateful information that is used to convert, parse and tokenise code element strings. On a host system, a text context object is of type **mbstate_t**.

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.

white space

A sequence of one or more characters that belong to the **space** character class as defined by the LC_CTYPE category in the current locale.

In the POSIX locale, white space consists of one or more blank characters (space and tab characters), newline characters, carriage-return characters, form-feed characters and vertical-tab characters.

wide-character code

An integer value corresponding to a single graphic symbol or control code. The object is of type **wchar_t** that corresponds to a member of the codeset of the locale on a host system.

wide-character string

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

Index