For: gwm
Printed on: Mon, May 2, 1994 19:02:40
From book: Arch-spec
Document: title
Last saved on: Mon, Sep 13, 1993 16:33:03
Document: Contents
Last saved on: Mon, Sep 13, 1993 16:34:21
Document: concepts
Last saved on: Sat, Sep 4, 1993 12:08:48
Document: techref1
Last saved on: Thu, Aug 12, 1993 21:36:17
Document: apiLocale
Last saved on: Mon, Aug 23, 1993 15:14:50
Document: apiLayout
Last saved on: Wed, Aug 18, 1993 18:10:25
Document: apiConv
Last saved on: Fri, Oct 22, 1993 17:26:18
Document: utilities
Last saved on: Thu, Aug 12, 1993 12:13:05
Document: features
Last saved on: Tue, Aug 17, 1993 14:29:56
Document: ucsover
Last saved on: Thu, Aug 12, 1993 22:41:40
Document: WrtSystem
Last saved on: Tue, Aug 17, 1993 14:18:56
(...)

Universal Language Support
*** Draft 1.2 ***
8/18/93+

Architecture Specification and C–Language Interfaces

Copyright 1993 by International Business Machines Corporation

Permission to use, copy and distribute this documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice and this permission notice appear in supporting documentation, and that the name of IBM not be used in advertising or publicity pertaining to distribution of the documentation without specific, written prior permission. IBM makes no representations about the suitability of this documentation for any purpose. It is provided “as is” without express or implied warranty.

Views expressed in this document are those of the authors, not of IBM.

IBM, Austin

Veronica Bonebrake – bonebr@ausvm1.vnet.ibm.com

Ian Holland – hollandi@bcrvmpc1.ibm.com

Gary Miller – gwm@austin.ibm.com

Frank Rojas – fxrojas@nlsarch.austin.ibm.com

Joe Ross – ross@austin.ibm.com

Section I: ULS Key Concepts **11**

1 Introduction 11
1.1 Internationalization problems 11
1.2 Universal Solution 12

2 Background 13
2.1 Scripts 13
2.2 What is UCS? 13
2.3 What is UTF−2? 13
2.4 ULS and Standards 13
2.4.1 Universal Code Set 13
2.4.2 Code Set Independence 14
2.5 Locales 14
2.5.1 National Locale 15
2.5.2 UNIVERSAL Locale 15
2.6 Levels of Language Support 15
2.7 Localized Objects and Subsystems 16

3 Universal Language Support 17
3.1 Character Type 17
3.2 Locale Object 17
3.3 Overview of Universal Functions 18
3.4 Universal Programming Model 18
3.5 Universal Subsystems 19
3.6 Coexistence with other codes 20

4 International Language Support 22
4.1 Character Type 22
4.2 UTF as a Bridge to Universal Character Set 23
4.3 Overview of Internationalization Functions 23
4.4 Internationalization Model 23
4.5 International Subsystems 25

Section II: ULS Technical Reference **27**

5 Introduction 27
5.1 Objectives 27
5.2 Non−objectives 27
5.3 Assumptions 27

6 Locale Functions 28
6.1 Locale Management Functions 28
6.1.1 UniCreateLocaleObject() − Create a locale object 28
6.1.2 UniQueryLocaleObject() − Retrieve the locale name 30
6.1.3 UniFreeLocaleObject() − Free a locale object 32
6.1.4 UniLocaleStrToToken() − Locale specification string to token 32
6.1.5 UniLocaleTokenToStr() − Locale token to locale string 33
6.1.6 Locale Token Format 33
6.2 Locale Information Functions 34
6.2.1 UniQueryLocaleInfo() − Retrieve locale conventions information 34

6.2.2 UniFreeLocaleInfo() – Free UniLconv structure 35
6.2.3 UniQueryLocaleItem() – Retrieve locale information by item 36
6.3 Date and Time 38
6.3.1 UniStrftime() – Format date and time 38
6.3.2 UniStrptime() – Date and time conversion 38
6.4 Monetary Formatting 39
6.4.1 UniStrfmon() – Convert monetary value to string 39

7 String/Character Functions 40
7.1 String Manipulation Functions 40
7.1.1 UniStrcpy() – Copy code element string 40
7.1.2 UniStrncpy() – Copy a specific number of code elements 40
7.1.3 UniStrcat() – Concatenate code element strings 40
7.1.4 UniStrncat() – Concatenate a specific number of code elements 41
7.1.5 UniStrcmp() – Compare code element strings 41
7.1.6 UniStrncmp() – Compare a specific number of code elements 41
7.2 String Search Functions 42
7.2.1 UniStrchr() – Search for the first occurrence of a code element. 42
7.2.2 UniStrcspn() – Search for code element string 42
7.2.3 UniStrpbrk() – Locate code elements in a code element string 42
7.2.4 UniStrrchr() – Locate last occurrence of code element 42
7.2.5 UniStrspn() – Determine the number of code elements in a segment . 43
7.2.6 UniStrstr() – Locate a code element sequence 43
7.2.7 UniStrlen() – Determine code element count 43
7.3 Composite Character Sequence (CCS) Functions 44
7.3.1 UniQueryCompCharLen() – Composite character sequence code element count . . 44
7.3.2 UniNextCompChar() – Advance to next composite character sequence 44
7.3.3 UniQueryCompChar() – Number of composite character sequences . 45
7.3.4 UniQueryCompStrDWidth() – Display width of a composite character sequence string 46
7.4 Character Attribute Functions 47
7.4.1 UniCreateAttrObject() – Initialize for attribute query 47
7.4.2 UniQueryCharAttr() – Query character attribute 49
7.4.3 UniScanForAttr() – Scan for attribute match 50
7.4.4 UniFreeAttrObject() – Release character attribute object 51
7.4.5 Atomic Query Attributes 51
7.5 String Transformation Functions 52
7.5.1 UniCreateTransformObject() – Initialize string transformation 52
7.5.2 UniTransformStr() – String transform 54
7.5.3 UniFreeTransformObject() – Free string transform object 54
7.5.4 Atomic Transformation Functions 55
7.6 String Conversion Functions 56
7.6.1 UniStrtod() – Convert character string to double precision floating point 56
7.6.2 UniStrtol() – Convert a character string to a long integer 57
7.6.3 UniStrtoul() – Convert a character string to an unsigned long integer . 58
7.7 String Comparison Functions 59
7.7.1 UniStrcoll() – Language collation string comparison 59
7.7.2 UniStrxfrm() – Transform a character string 59
7.8 Character Set ID Function 60

7.8.1 UniQueryCharSetId() − Determine the CharsetId of character 60

8 Layout Functions 61
8.1 Layout Text Functions 61
8.1.1 UniCreateLayoutObject() − Initialize a layout context 63
8.1.2 UniFreeLayoutObject() − Free a LayoutObject 63
8.1.3 UniSetLayoutValues() − Set Layout Values of a LayoutObject 64
8.1.4 UniGetLayoutValues() − Get Layout Values of a LayoutObject 65
8.1.5 UniTransformLayout() − Layout transformation 66
8.1.6 UniEditShapeLayout() − Shape Editing of context dependent text 69
8.1.7 Layout Values 71
8.1.8 Layout Value Types 71
8.1.9 ActiveBidirection 72
8.1.10 ActiveShapeEditing 72
8.1.11 UniShapeCharset 72
8.1.12 ShapeCharsetSize 73
8.1.13 ShapeContextSize 73
8.1.14 Orientation 73
8.1.15 TypeOfText 73
8.1.16 BidiType 74
8.1.17 Swapping 74
8.1.18 Numerals 75
8.1.19 Text Shaping 75
8.1.20 ArabicSpecialShaping 75
8.1.21 ArabicOneCellShaping 76
8.1.22 WordBreak 76

9 Conversion Functions 77
9.1 Conversion Overview 77
9.2 Universal Conversion Functions (Uconv) 78
9.2.1 UniCreateUconvObject() − Open and initialize a Uconv object 78
9.2.2 UniUconvToUcs() − Convert from Multibyte to UCS 79
9.2.3 UniUconvFromUcs() − Convert UCS to Multibyte 81
9.2.4 UniFreeUconvObject() − Close and free memory of a UCS conversions object 83
9.3 International Conversion Functions (ICONV) 84
9.3.1 iconv_open() − code conversion allocation function 84
9.3.2 iconv() − Perform code set conversion 85
9.3.3 iconv_close() − Close and free memory of a conversion descriptor ... 87
9.3.4 iconv_query() − Query for possible conversions 88
9.4 Conversion Modifiers 89
9.4.1 Locale Argument 90
9.4.2 Substitute Argument 91
9.4.3 Substitute Character Argument 91
9.4.4 Orientation Argument 91
9.4.5 Type of Text Argument 91
9.4.6 Swapping Argument 92
9.4.7 Numerals Argument 92
9.4.8 Text Shaping Argument 92
9.4.9 Endian Argument 93
9.5 Managing Coded Character Set Identifiers For Interoperability 94

9.5.1 List of Supported Tag Types 94
9.5.2 UniUcsidToTag() − Map UCSID to identifier 96
9.5.3 UniUcsidFromTag() − Map UCSID from identifier 97

10 Commands and Utilities 98
10.1 Locale management 98
10.1.1 localedef 98
10.1.1.1 Charmap Files 98
10.1.1.2 Locale Source File 103
10.1.1.3 Methods File 113
10.2 Layout management 117
10.2.1 layoutdef 117
10.2.1.1 Layout Source File 117
10.2.1.2 LayoutMethods File 121
10.3 Conversion 123
10.3.1 iconv Command 123
10.3.2 uconvdef Command 124
10.3.2.1 Uconv Source File 124

ULS Localization 128

11 Locales 128
11.1 UNIVERSAL Locale 128
11.2 Localized UCS Locales 128

12 Conversion 130
12.1 Code Set Naming Conventions 130
12.2 UCS−2 Conversion Support 130
12.2.1 Minimum Support 130
12.2.2 PC Interoperability Support 130
12.2.3 Lists of UCS−2 Interchange Converters 130
12.2.3.1 Standard and Open Systems UCS−2 Converters 131
12.2.3.2 PC UCS−2 Converters 132
12.2.3.3 EBCDIC UCS−2 Converters 133
12.3 Additional Conversion Support 133
12.3.1 Lists of UTF−2 Interchange Converters 134
12.3.1.1 Standard and Open Systems UTF−2 Converters 134
12.3.1.2 PC UTF−2 Converters 135
12.3.1.3 EBCDIC UTF−2 Converters 136
12.3.2 List of Converters for Standard, PC and EBCDIC Regional Code Sets 136

Appendix A: Unicode / ISO10646 Overview 138

11 UCS Architecture 138
11.1 Coded Representation Forms 138
11.2 Implementation Levels 139
11.3 Subsets 139
11.4 ISO 10646 Repertoire and Allocation 139
11.5 UCS Transformation Format (UTF−1) 140

12 UTF−2 − A new UCS Transformation Method Proposal 140
12.1 The UTF−2 Definition: 141

13 Unicode 141
13.1 Scope 141
13.2 The Unicode Consortium 142
13.3 Unicode Availability 142
13.4 Unicode 1.01 and Unicode 1.1 142

14 Unicode and ISO 10646 Scope Differences 143
14.1 Character Attributes 143
14.2 Bidirectional Architecture 143
14.3 Conformance 143

15 What does it mean to implement Unicode 144
15.1 Unicode as interchange code 144
15.1.1 Benefits 144
15.1.2 Costs 145
15.2 Unicode as a processing code 145

Appendix B: Writing System Analysis 146

Appendix C: ULS Programming Tips 150

16 XPG/ISO−C vs ULS Function Comparison 150

17 UCS Processing 151

18 UCS Intechange 151

19 UCS Presentation / Input 151

20 UCS File Storage 151

Appendix D: ULS Input and Output 152

21 Text Data Object and I/O Functions 152
21.1 Text Data Object Functions 152
21.1.1 UniCreateTextDataObject() − Create a text data object 153
21.1.2 UniQueryTextDataObjectLocale() − Query text data object locale data 154
21.1.3 UniQueryTextDataObjectStream() − Query text data object stream data 154
21.1.4 UniQueryTextDataObjectConv() − Query text data object conversion data 155
21.1.5 UniFreeTextDataObject() − Free text data object 155
21.2 Input/Output Functions 156
21.2.1 UniInput() − Input character string 156
21.2.2 UniOutput() − Output a character string 157
21.2.3 UniUnGet() − Push a character onto a stream 158
21.3 Formatted Input/Output Functions 159

21.3.1 UniFprintf() – Write a formatted string to a stream 159
21.3.2 UniSprintf() – Write a formatted string to memory 162
21.3.3 UniFscanf() – Read a formatted string from a stream 163
21.3.4 UniSscanf() – Read a formatted string from memory 166

Glossary 167

Bibliography 180

Trademarks

Unicode is a trademark of the Unicode Consortium

X/Open and XPG are trademarks of X/Open Company Limited

All other company and product names are trademarks or registered trademarks of the company or manufacturer respectively.

Acknowledgements

A special thanks goes to Isai Scheinberg for his contribution to the ISO 10646, the Unicode Consortium, and the initial prototype of the Unicode Toolkit 1.0. His persistence and dedication to the universal character set movement made things happen.

Our thanks also go to the authors of the Unicode Toolkit 1.0 specification which has helped to crystalize some of the UCS issues within IBM. These include Mati Allouche, Milos Lalovic and Mel Sher.

Document Guide

This document is divided into 3 major areas and several appendices. The following documents are suggested reading for those who are unfamiliar with Unicode or ISO 10646 :

- ***The Unicode Standard, Worldwide Character Encoding, Version 1.0, Volume 1.***
- ***ISO/IEC:..., Information Processing – Universal Coded Character Set (UCS) – Part 1: Architecture and Base Multilingual Plane.***

This document describes the functions, utilities and commands that are expected to be provided by the **Universal Language Support** package. The following is a list of the major sections that will be common to all implementations of **ULS**:

Section I: ULS Key Concepts

This provides a quick introduction to the main concepts needed prior to reading the reference section. A programming model is presented in this chapter.

Section II: ULS Technical Reference

This section describes the functions, utilities and commands contained within the **ULS** environment. The programming interfaces described in this section are specified using the **C** language.

In addition, appendices are provided to give additional information:

Appendix A: Unicode/ISO 10646 Overview

This appendix provides a quick introduction and discussion of each of the Unicode and ISO 10646 specifications. The relation between these two is also provided.

Appendix B: Writing System Analysis

This appendix provides an analysis of the world's writing systems and identifies the main features of written languages that a computing system should support.

Appendix C: ULS Programming Tips

This appendix provides a set of tips on how the **ULS** package can be used.

Appendix D: ULS Input and Output

This appendix provides a description of a set of possible **ULS** I/O functions.

More information may be obtained from other publications listed in the **Bibliography** section of this document.

Audience

This document is intended for the following audience:

Planners	Planners of products that intend to use Unicode. Such planners need the information about what **UCS** support is expected. Particularly appropriate are the section on Concepts and Appendix A: Unicode/ISO 10646 Overview.
Developers	Developers of internationalized services expecting to support Unicode.
Localization	Developers intending to develop localization support for their national region may want to read this document to understand the relationship between their national code set support and the **ULS.**

Section I: ULS Key Concepts

1 Introduction

In this chapter, key concepts of Universal Language Support (**ULS**) are introduced.

The approval of the ISO 10646 standard and the activities of the Unicode Consortium have accelerated the recent interest in universal character sets. At the same time, the efforts of POSIX, X/Open and X Windows in standardizing the set of internationalization interfaces have achieved a new level of consistency in today's global open market.

1.1 Internationalization problems

Some of the problems being faced are:

1. Hundreds of *different code sets* and encoding schemes are used to encode character data. Some of these codes/interfaces are optimized for a particular national language or to a regional group, others are optimized for a particular application or system, and still others are there for historical reasons. This multiplicity of codes is a nightmare for any developer who has to implement products for today's global open market.

2. Software developers are faced with trying to build a *single* product for the global market; they must address the language and cultural requirements of various regions. Developers have long since learned the lesson that rewriting programs to make them work with new languages is wasteful, so they use system functions for sorting, displaying messages, formatting the date and time of day, and separating text from code. But today applications need to be developed such that they *do not require rewriting* for new script processing from various cultures.

3. Demand is increasing for *multilingual* applications that allow users to switch, not only from one language to another within a script, but also from one language and script to another language and script. The key to multilingual script processing services is understanding the *context* of every character in a string, and understanding that there are languages where the context of a character depends on its surrounding characters. For applications to handle various scripts successfully, they must process strings in groups of a size sufficiently large for the parser to determine the context of each character.

4. The *char* type within the C Programming Language has been established to be 8 bits wide thus allowing a total number of 256 individual characters to be represented. The problem is that there are more than 256 character glyphs commonly used in the world today. To overcome this limitation, code pages (sometimes called code sets) were introduced to define separate assignments of character shapes required for a particular language in some region of the world to the available char 8−bit values. Therefore, a char value (for example, decimal numeric value 210) will denote a different character shape depending on the code set selected.

5. In many countries, including Western nations, languages are based on relatively small inventories of about 94−128 characters. There exists a code set that can define an encoding of all the characters and symbols of such languages. Each individual character of the language will be encoded by a single byte value defined by the code set. However, problems arise if a user wants to use more than one language at a time, or if the user needs to transfer text internationally. The sender and receiver must agree on the code set used to encode the transferred text.

6. Another major problem occurs when we consider the Asian countries, such as Japan, Taiwan, China, or Korea. These countries have scripts that are based on conceptual pictures called logograms. A logogram is a character that represents a complete word or morph. (Sometimes such characters are called ideographs; however, ideograph is considered by many scholars to be an inaccurate term because it implies that the character can represent an entire idea or concept. An entire concept can also be

represented in a phrase, sentence, paragraph, or discourse.) A complete collection of logograms for the languages of these countries contains many thousands of characters. Therefore, any representation of East Asian text must use more than one byte to denote uniquely a particular character. Asian C programmers continue to use the char datatype, but are aware that an individual logographic character will be encoded by possibly two or more char values.

7. To further complicate this problem, modern national character sets incorporate English (Latin−based) words, digits or symbols because Latin countries have been strongly influenced by Western culture and technology. In addition, computers did not initially support many languages: early users were forced to use single−octet systems and applications. The result of this situation is that the software system's representation of text must support the combination of a few logograms and a Latin alphabet, whereas existing multibyte code pages support a single language plus English. This small inventory of characters is too limited for today's global market.

8. Several systems have used and are still using stateful encodings of coded characters while other systems reject such encodings. The most common existence of stateful encodings are the Asian EBCDIC encodings that use SI/SO (Shift in/Shift out control codes) to change between single byte and double byte encodings. The main issue is that the industry has not settled on a) whether this is justified and b) which encoding method to use. The current standards (ISO−C, XPG, POSIX) make it very difficult to support both stateful and stateless encodings in a portable manner.

9. Many scripts of the world require context−dependent drawing where the shape of a given character depends on the surrounding characters. As yet the internationalization standards have not defined how such scripts should be handled.

1.2 Universal Solution

The Universal Language Support (**ULS**) specification describes a set of functions and utilities for addressing the problems listed above by providing:

- A *Universal Coded Character Set* capable of encoding most of today's languages and scripts.
- A set of *Universal Language Support* functions for the processing, input and output of the above.
- A set of *Universal Locale* objects that describes processing of text on a global scope.
- A set of *Universal Layout* objects that describes processing of text on a global scope.
- A set of *Universal Conversion* objects for import/export of traditional data into the **ULS** environment.

This specification identifies the level of support expected in all future operating systems for support of *Universal Coded Character Sets*. Without abandoning its standards−based internationalization support, **ULS** will add support for a set of *UCS locales* as extension to the existing internationalization language support.

2 Background

The purpose of this chapter is to provide a background on the high−level concepts that the **ULS** will work with.

2.1 Scripts

Today's graphical user interfaces allow very rich text processing to be part of most applications and this requires systems to handle scripts from around the world. The following lists the types of scripts and some of the languages that use them.

Latin type	Western/Eastern European languages, Russian, ...
Logographic type	Japanese, Chinese, Korean, ...
BiDirectional type	Arabic,Hebrew, ...
Indic type	Thai, Devanagari, ...

Traditionally, applications are built for only one of these scripts. The purpose of listing these scripts is to emphasize that **ULS** embraces the Unicode concepts and aims to support all of these scripts even in its early stages. Further detail on types of scripts is provided in Appendix B.

2.2 What is UCS?

UCS *or "Universal Coded Character Set"* is a new global character encoding, which was developed jointly by the industry and the International Organization for Standardization (ISO), and defines the new state of the art for character handling. The industry name for this 16−bit code is **Unicode** while the ISO 10646 name is *"Universal Multiple−Octet Coded Character Set"*. ISO 10646 allows either a 16−bit code (**UCS−2**) or a 32−bit code (**UCS−4**) to be used. However, the 16−bit code is already establishing itself as the predominant code being used, while the 32−bit code is of limited practical interest. The focus of this document is based on the development of Unicode applications, and the terms **UCS−2** and **Unicode** will be used interchangeably. For a more detailed description of the universal character set refer to **"Unicode** / **ISO 10646 Overview"** (Appendix A).

2.3 What is UTF−2?

UTF−2 refers to the new *UCS Transformation Format* developed by the X/Open internationalization working group. It is an alternative encoding of **UCS** that is compatible with traditional ASCII−based systems and is assured to be "file system safe" for PC and workstations. **UTF−2** is a multibyte code set capable of encoding the same set of characters of **UCS−2** in 1−3 bytes or **UCS−4** in 1−6 bytes but using a scheme for which

− ASCII (7−bit code) is a proper subset
− preserves the semantics of the portable character set
− preserves the semantics of a null octet for C Programming Language.

It is expected that **UTF−2** will be a dominant transformation method where **UCS** is not practical. For a more detailed description of **UTF−2** refer to **"Unicode** / **ISO 10646 Overview"** (Appendix A).

2.4 ULS and Standards

Internationalization refers to the process of developing applications that can modify their behavior at run time depending on the language and cultural needs of the end user. This requires an internationalized program to make no assumption about the language of the character data (text) that it is designed to handle. Two approaches are identified for achieving this goal.

2.4.1 Universal Code Set

The Universal Language Support allows development of internationalized applications by providing similar functions to the ISO−C and XPG for accessing cultural information but uses **UCS** for all text processing. In addition, the **ULS** provides functionality that allows handling of BiDirectional and Indic scripts in a uniform

fashion that is based on the principles of Unicode. It is left up to the application writer to decide whether the application is to store the UCS data directly onto the file space, or to convert it to some national code set. The following illustrates an internationalized application whose text processing is fully aware of the **UCS** encoding.

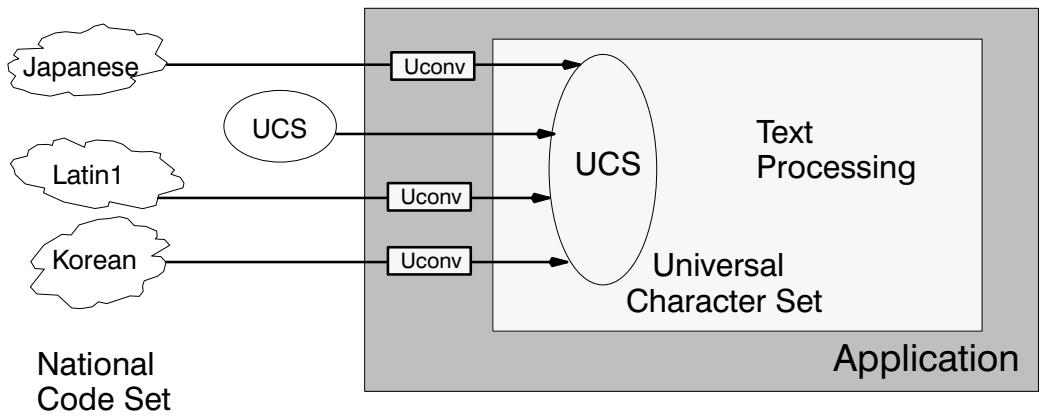


Figure 1. Universal Code Set Approach

2.4.2 Code Set Independence

The evolution of ISO−C and XPG standards for development of international applications is based on the concept of *code set independence* in which the encoding of the data is hidden from the application. Interfaces are provided that allow parsing, collation, and formatting of multibyte text without knowing the specific code set. The ISO C standard defines a **wide character** data type that allows text processing using a uniform size but whose encoding and size is unknown for any given implementation. These standard interfaces do allow an application to be internationized and do allow it to process data encoded in various code sets. These interfaces can only support scripts of type *Latin* and/or *Logographic*. But **UCS** can not be used as an encoding for wchar_t in all cases (EBCDIC). The following illustrates a code set independent application that converts the user's data (National Code Set) into a normalized (wchar_t) form. The text processing does not really know the wchar_t encoding.

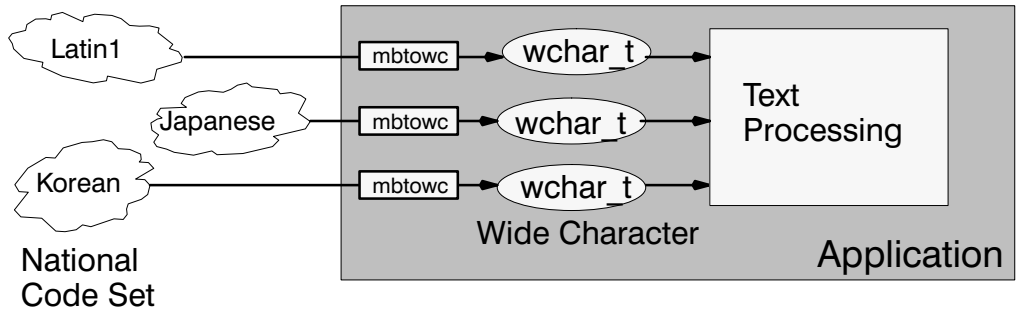


Figure 2. Code Set Independence Approach

2.5 Locales

Internationalized applications will be required to operate in a variety of environments based on some territory, language and/or cultural definition. These environments are identified by a *locale* which is simply a name. In addition, a locale name is used to identify the set of *localized objects* that encapsulate the information for processing the associated environment. Each localized object is named according to its type and the locale name. Remember that the locale identifies the culture, script, language, and territory that it supports. Within the **XPG** definition of locale names a code set may also be considered part of the locale name but within **ULS** locale names will not include any code set name.

The XPG format of a locale name is based on the X/Open Portability Guide (XPG) definition, as follows:

```
language[_territory][.code set][@modifier]
```

As a convention, there will be a name for **UCS** locales without the code set name.

The language and territory abbreviations are based on the **ISO 639** and **ISO 3166** standards for language and territory representation. The modifier is not yet defined, but all implementations shall allow future use of the modifier. For example, the locale for Danish as spoken in Denmark is **da_DK**, where **da** stands for the Danish language, **DK** stands for Denmark.

An application will use the **UniCreateLocaleObject()** function to obtain a handle to the locale object requested by the user.

2.5.1 National Locale

A set of national locales will be provided by **ULS**. An internationalized application may be required to display information differently in different locations. For example, the convention for date and time formats is different in the United States than in England. Similarly, the decimal point (radix character) used for formatting numeric information, the affirmative and negative response strings, the monetary symbols and formatting of monetary and numeric quantities are different across the countries.

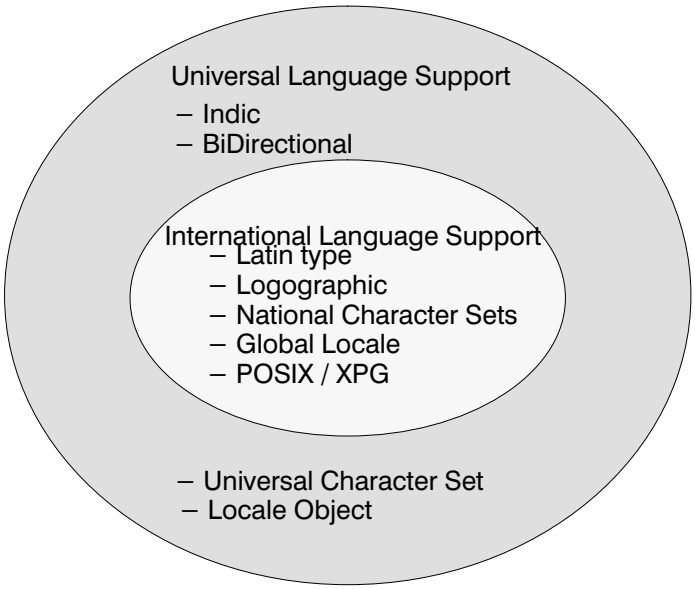
2.5.2 UNIVERSAL Locale

A special **UNIVERSAL** locale will be provided. This locale will include the language−independent character attribute definitions detailed by the **Unicode** specification ([13], [14]). This set of generic attributes will be defined in a source file that can be merged with other national locales to produce a ”mixed environment” in which certain generic character attributes (e.g. graphic or printable) are derived from the **Unicode** specification, but certain national locale components (e.g. date, time and monetary formats) are specific to a particular language and/or territory.

2.6 Levels of Language Support

In order to provide the flexibility to support multiple code sets and encodings, and to allow users to add new locales, **ULS** provides an architecture that uses object−oriented principles but is developed using the **C** programming language. The term *Language Support* is used to refer to the functions, libraries and utilities provided to support the interfaces defined in Section II. The goal of the Language Support is to establish the set of subsystems that enables dynamic customization and at the same time provides a consistent set of *localized objects* (e.g. locales, conversions, fonts, keyboard maps) that will be common across any platform it is ported to. All subsystems are designed to be platform−independent and to be highly portable across platforms that support the C Programming Language.

The following is a view of the division of functionality provided by the various levels of language support:



The figure represents the division between the International Language Support and the Universal Language Support. The scope of this document focuses on the Universal Language Support, which provides the capability of incorporating Indic and BiDirectional support. The International Language Support (I18N) is based on ISO−C and XPG4 standards.

2.7 *Localized Objects and Subsystems*

Under the **ULS** architecture, *localized objects* refer to opaque structures that define access to locale methods, conversion methods, input methods or catalogs of messages. For each localized object, the **ULS** will bundle the data and functions associated with each of these localized objects into a file that can be dynamically loaded on demand. Localized objects are associated with a specific *subsystem* that defines internal functions to be used to meet the public interfaces defined in the **ULS** specification.

Not all subsystems are well−defined at this time; the following provides an overview of each subsystem.

Locale Subsystem

Set of structures, tables and functions to implement the culture−specific **ULS** public interfaces. All methods within the locale subsystem will use UCS−2 as the internal process code.

Layout Subsystem

Set of structures, tables and functions to implement the layout specific **ULS** public interfaces. All methods within the layout subsystem will use UCS−2 as the internal process code.

Conversion Subsystem

Set of structures, tables, methods, and functions for each conversion provided by **ULS**. The conversion subsystem will define the Uconv and iconv methods. Universal conversions will be expected to handle **UCS** as either octet encoded or as UCS−2 (16−bit integral) value.

3 Universal Language Support

3.1 Character Type

Universal code (UniChar)*

The universal code (**UCS**) can be used to represent more characters than can be represented using any current national or regional coded character set. In addition, **Unicode** specifies a consistent and uniform set of rules for text processing that define how implementations should behave with respect to contextual processing and combining characters.

There is no standard that defines a unique data type for **UCS** code. The **ULS** defines the *universal character* type

UniChar

which is a typedef of a 16−bit integral type whose range of values can represent distinct codes for all members of the Basic Multilingual Plane of ISO 10646. The null character shall have the code value zero, and each member of the Portable Character Set as defined by POSIX 1003.2 shall have a numeric value equal to the value defined by ISO 646:1991 IRV.

Within the **ULS**, applications can assume that the ***UniChar*** data type is a 16−bit integer type. Furthermore, any function whose prototype includes an argument of type **UniChar*** may be an implementation conforming to ISO 10646 definition of Level−3 (combining) with no subsets. The Level−3 implementation defines that multiple code elements (**UniChar**) can be combined to map to a single character shape (graphic symbol). As such, a single character shape (graphic symbol) could have two or more different code representations. It is this combining characteristic that distinguishes **UCS** from a process code. Applications should strive to conform to Level−3 definition wherever possible.

All text processing interfaces defined in the **ULS** use the **UniChar** type for representation of characters and it is recommended that **UniChar** be used for text processing in this programming model. The **UniChar** may be a typedef of the **wchar_t** data type when the latter is defined as a 16−bit integral data type. Using **UniChar** will enable applications to be portable to platforms where the **wchar_t** is not defined to be 16−bits.

3.2 Locale Object

The **ULS** functions are expected to be used in programming environments requiring multiple different locale contexts for processing of language−specific data. Such environments include object−oriented, multi−threaded, graphical, or distributed networking contexts.

There is no standard that defines a type for managing locale−dependent contexts. The **ULS** defines the *locale object*

LocaleObject

which is an opaque data type that encapsulates the locale−dependent data and functions. Any **ULS** function that depends on a locale will take a **LocaleObject** argument which must be allocated by the application. The **ULS** functions are said to be *non−global* since their behavior is independent of the global locale controlled by the **setlocale**() function.

3.3 Overview of Universal Functions

A full set of functions is defined such that a program can process character strings using **UniChar** entirely. For example, the **UniStrlen()** function determines the length of a universal string.

Some functions whose signature includes a single **UniChar** argument, e.g. **UniQueryCharAttr()**, are not capable of supporting the Level 3 (combining) aspect of **UCS** codes. An example is determining whether a character has the **alpha** attribute. With Level 3, the question is whether the combining character on its own has the **alpha** attribute. To avoid such issues these functions should be avoided if the application intends to conform to the definition of Level 3. Any function whose signature is defined to take a **UCS** string (**UniChar*)** will not have this limitation.

3.4 Universal Programming Model

To support multiple languages, the following items form the basis for the universal programming model. These items will also highlight the changes needed to make an existing or new program able to support multiple languages.

- Provide support for any language. Do not assume any specific subset of languages. Do not assume that characters or text strings have any specific properties; let the application determine them dynamically using the appropriate ***UniChar*** interfaces.
- Programs that need to support multiple languages need not be concerned with a specific national code set because they use **UCS** for all internal processing. Such programs must be able to import and export user data into/from environments that use national code sets. Programs may assume that all national characters are encoded using **UCS**.
- **ULS** does not provide any tagging services other than providing identification of **UCSID** (refer to "Managing Coded Character Set Identifiers for Interoperability"). It is left to the application to determine the tagging context for using the **ULS** functions and to decide how text strings will be associated with any tag including **UCSID**.
- Programs should use **UTF−2** to export **UCS** to an ASCII−based system when the user's preferred code set cannot be determined. System utilities may not be able to recognize characters not included in the the Portable Character Set. (See Appendix A section 14 for further information.) The **UTF−2** can be used when information needs to be preserved.
- A library that claims to have **ULS** should support interfaces based on ***UniChar*** or ***UniChar**** data types. Any language−sensitive processing is encapsulated into a language object with a set of methods that can be used to manipulate text strings.
- To handle multiple languages, the non−global functions should be used to handle switching between various language objects.

Figure 3 illustrates the flow of data between applications, the ULS and system components.

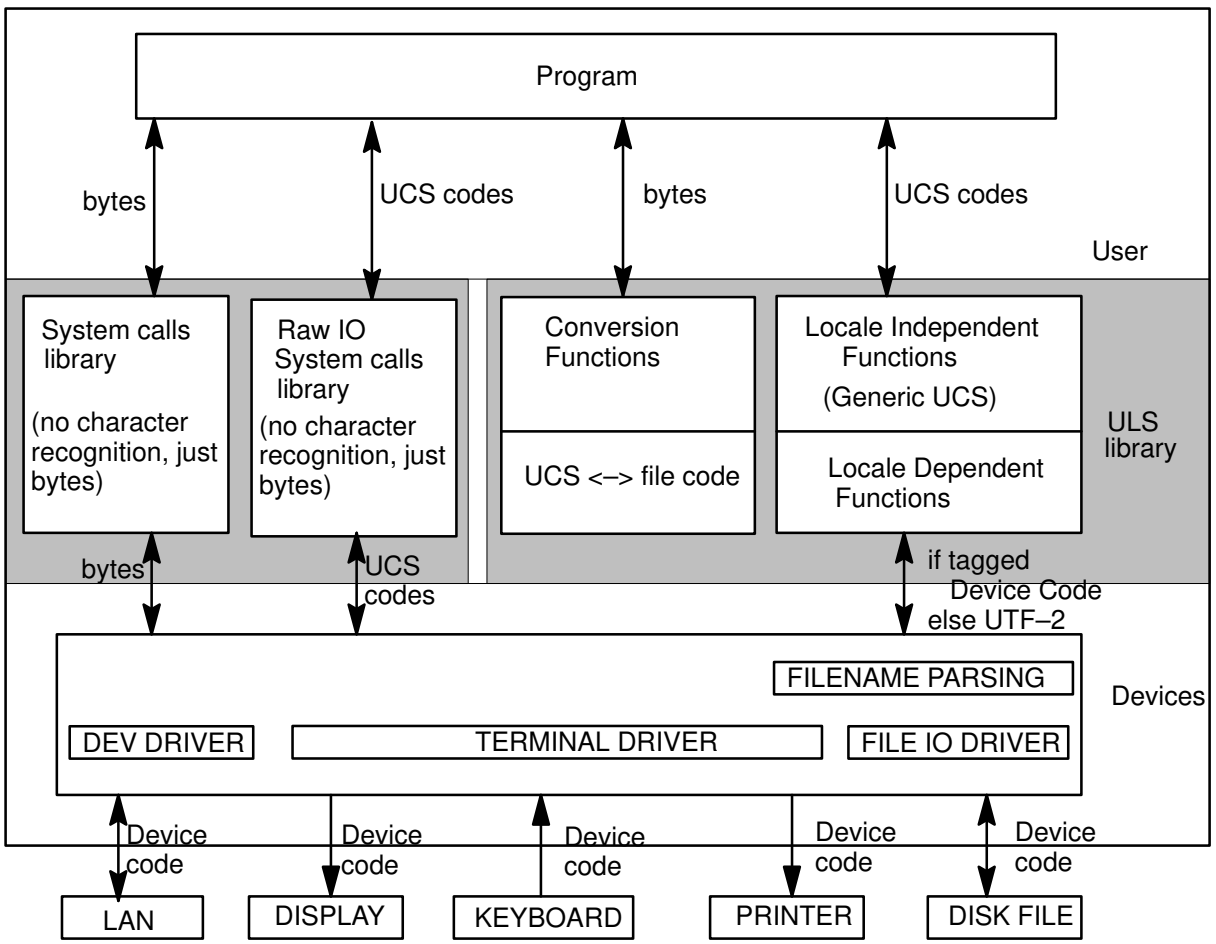


Figure 3. Universal Programming Model

3.5 Universal Subsystems

The following shows an illustration of the **ULS** subsystems and the public interfaces that can be expected to be found within a **ULS** offering.

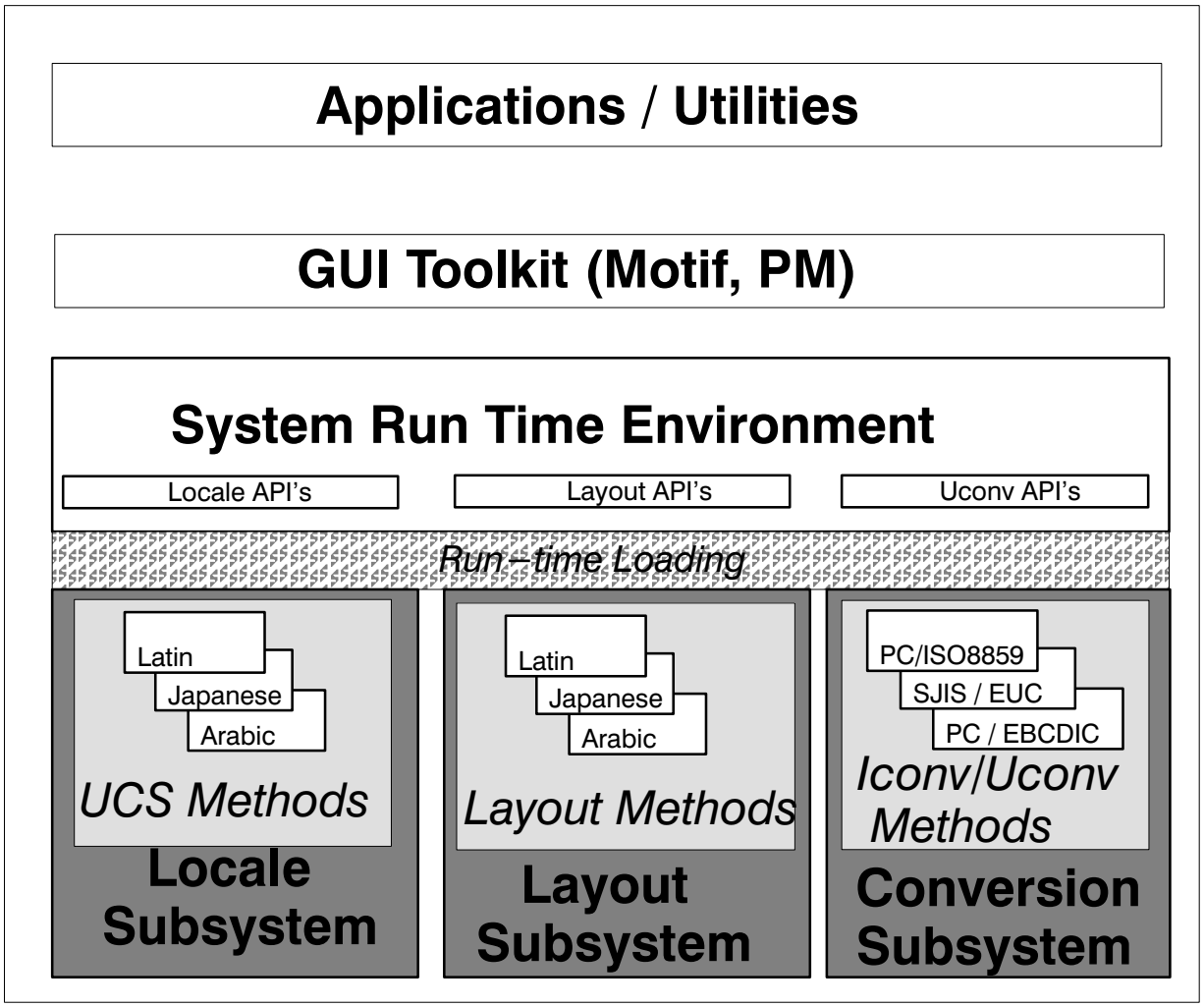


Figure 4. ULS Programming model for internationalization of applications

3.6 Coexistence with other codes

Passing of **UCS** data to applications that have not been enabled for the universal character set may cause interoperability problems because the encoding of **UCS** uses null octets. For this reason, the following recommendations should be followed when exporting **UCS** data to a non−**ULS** environment:

- First try to determine the national code set the user desires.
- If a national code set cannot be determined and the application must output **UCS** data in an ASCII−based environment, convert UCS−2 to UTF−2.
- If a national code set cannot be determined and the application must output **UCS** data in an EBCDIC−based environment, convert UCS−2 to IBM−500.

Applications that expect to use **char*** utilities and libraries will need to convert strings to/from its **UniChar** form and invoke the appropriate functions in some cases. The only mechanism to convert from a char* to a **UniChar*** is the **Uconv** interface. The application writer is responsible for the application to determine the appropriate national code set being used by a user. The following are some possible scenarios for determining the national code set:

1. Application can call nl_langinfo(CODESET) to get the global locale's code set.

2. Application can query some system or user configuration database for the national code set.
3. Application can prompt the user to choose from a list of possible candidates.
4. Application may receive a tag that identifies the national code set.

4 International Language Support

This section describes the codeset−independent, **XPG** programming model. Applications which are internationalized in this way can use the **ULS** locales to provide users with multiscripting capability. Such applications will not be able to make use of the **UCS**−specific interfaces of the **ULS**. Software developers interested only in explicit **UCS** processing should skip this section.

4.1 Character Type

The following discusses the various data types that may be used to encode text data. The choice of data type will depend on the programming model used and the programming interfaces used within that programming model.

Multibyte codes (char)*

A multibyte code is the bit pattern representing a character in a code set. IBM−850 and IBM−932 are examples of **code sets** that define multibyte codes. Similarly, Japanese EUC, Korean EUC, PRC Chinese EUC, ROC Chinese EUC and Thai 8−bit encodings are some of the other Asian multibyte encoding schemes. To represent a large number of characters, the EUC encoding scheme uses multiple bytes. The multibyte code is the traditional external representation of text data based on some regional code set, be it a character input from a keyboard or a file on a disk. The number of bytes used to represent an individual character can vary within a single code set. One has to use a multibyte processing function on the first byte of the multibyte code of any character to determine how many bytes make up that character. For example, to obtain a character, based on the code set in use, all the bytes that make up the character need to be obtained.

Within workstation markets, the multibyte codes typically supported are said to be ”ASCII−based”, meaning that the code set contains the set of 7−bit ASCII (ISO−646) characters and that their coded values are based on the ASCII code.

Process code (wchar_t)*

The concept of a **process code** grew out of the necessity of processing multibyte characters internally in the system. The idea is to convert the external multibyte representation into a uniform internal representation, in which all characters have the same length. Using this internal form, character processing can be done in a code set independent fashion. This representation of characters is what is referred to as **process code**.

A **wide character code**, as defined by the ISO−C internationalization standards is the process code of a single coded character in a given locale. The ANSI C standard defines the type **wchar_t** to represent wide character code values. This type is defined by a **typedef** statement in the **<stddef.h>** and **<stdlib.h>** header files. The number of bytes required for a wchar_t value is not specified by the standards and is implementation−dependent. Programs expecting to be portable must not assume a particular size for wchar_t.

> **Note:** For XPG/ISO−C, functions provided by the **ULS** are internally associated with a locale context whose process code is encoded as **UCS**. Portable XPG applications should not depend on this but rather assume that the process code (**wchar_t**) encoding is unknown.

ULS will provide a set of *UCS locales* whose *process code* will be encoded using **UCS**. All *UCS locales* may be announced using the ANSI C function **setlocale**(). Programs that depend on the availability of the *UCS locales* will not be expected to be portable. It should be noted that methods of a *UCS locale* may be limited to level 1 **UCS**. Some systems support, for example, 64−bit or even 8−bit definitions of wchar_t in implementing the process−code model. Readers unfamiliar with the different levels of UCS can find them explained in the **Unicode** / **ISO 10646 Overview** discussion in Appendix A.

4.2 UTF as a Bridge to Universal Character Set

UCS provides the capability to encode multilingual text within a single coded character set. However, **UCS** and its **UTF−1** variant do not protect null bytes and/or the ASCII slash ("/") making these character encodings incompatible with existing Unix implementations.

The **UTF−2** provides an ASCII−compatible transformation format of **UCS** such that any traditional ASCII−based systems can support multilingual text in a single encoding. This transformation format encoding is intended to be used as a file code. This transformation format encoding of **UCS** is intended as an intermediate step towards full **UCS** support. However, since nearly all Unix implementations face the same obstacles in supporting UCS, **UTF−2** provides a common and compatible encoding during this transition stage. For this reason the code set (i.e. nl_langinfo(CODESET) of any *UCS locale* will be **UTF−2.**

4.3 Overview of Internationalization Functions

Because of the existence of two types of encodings −− multibyte and process −− there are two sets of functions for processing data from any multibyte code set.

ANSI (multibyte code) functions

Functions that process characters in multibyte code form are called multibyte functions. These usually start with the prefix **mb**. There are also some multibyte functions that do not have this prefix. For example, the string functions **strcoll()** and **strxfrm()** process characters in their multibyte form but do not have the **mb** prefix. Note that the standard C functions **strcmp()**, **strcpy()**, **strncmp(), strncpy(), strcat(), strncat()** that operate on bytes can be used in handling multibyte data. The standard C search functions **strchr()**, **strrchr()**, **strpbrk()**, **strcspn()**, **strrchr()**, **strspn()**, **strstr()**, **strtok()** can be used in the following cases:

- searching/scanning for characters in single byte code sets
- or, searching/scanning for 7bit ASCII in multibyte strings.

All multibyte functions are said to be global because their behavior depends on the global locale setting.

Wchar_t functions

Functions that process characters in **wchar_t** form are called wide character functions. Wide character functions usually start with a **wc** prefix. There are some wide character functions that do not have this prefix, as for example, the wide character classification functions (**iswalpha()**, etc.). An easy way to decide if a function is a wide character function is to check if the function prototype defines characters as a **wchar_t** data type or **wchar_t** data pointer or if it returns a **wchar_t** data type. There are some exceptions to this as, for example, the wide character classification functions which accept the wide character code cast as an integer value.

In addition, note that several **wchar_t** based functions are defined that do not have corresponding **char*** multibyte code based functions. Because of this asymmetry, it may be necessary in some cases to convert strings to its **wchar_t** form and invoke the appropriate functions.

Wide character functions may operate on either process code or UCS code. If a UCS locale is being used, wide character functions whose function prototype include a **wchar_t** argument, e.g. **iswalpha()**, will ignore the Level−2 and Level−3 (combining) aspect of UCS codes. Using such functions should be avoided if full UCS code support is intended. Wide character functions whose signature takes strings (**wchar_t*)** do not have this limitation.

All wide character functions are said to be global because their behavior depends on the global locale setting.

4.4 Internationalization Model

A programming model is presented here for internationalizing programs. This model should serve the purpose of highlighting changes when an existing program is internationalized and when a new program is developed:

- Provide complete internationalization. Do not assume that characters have any specific properties; determine them dynamically using the appropriate interfaces. Do not assume properties of code sets except those explained in this chapter about ASCII characters with code points in the range 0x00 through 0x3F (this range is referred to as the **unique code point range**).
- Make programs code set independent. Programs should not assume single, or double or multibyte encoding of any sort. The application can process data either in process code form or file code form using the appropriate ANSI C and X/Open XPG3 and XPG4 functions.
- Interaction with the kernel (e.g., file names) is provided only in the file code form. The kernel **does not handle process codes**.
- A library that claims to have international language support should provide all the functions defined in the standards. Any language−sensitive processing is encapsulated into a language object with a set of methods that can be used to manipulate text strings.
- Libraries may or may not provide processing in **wchar_t** format. An application that needs a library that does not support such a format must use file codes when invoking functions from such libraries.
- Programs can process characters either in the **wchar_t*** form or **char*** file code form. It is possible to write code set independent programs using both methods. Note, however, that there may be a performance penalty (depending on the program) for either approach and the better of the methods should be chosen on a per−program basis.
- Use non−global functions in cases where a unique language object context is required within a thread of operation, e.g. threaded servers, object−oriented environments and exception handling.

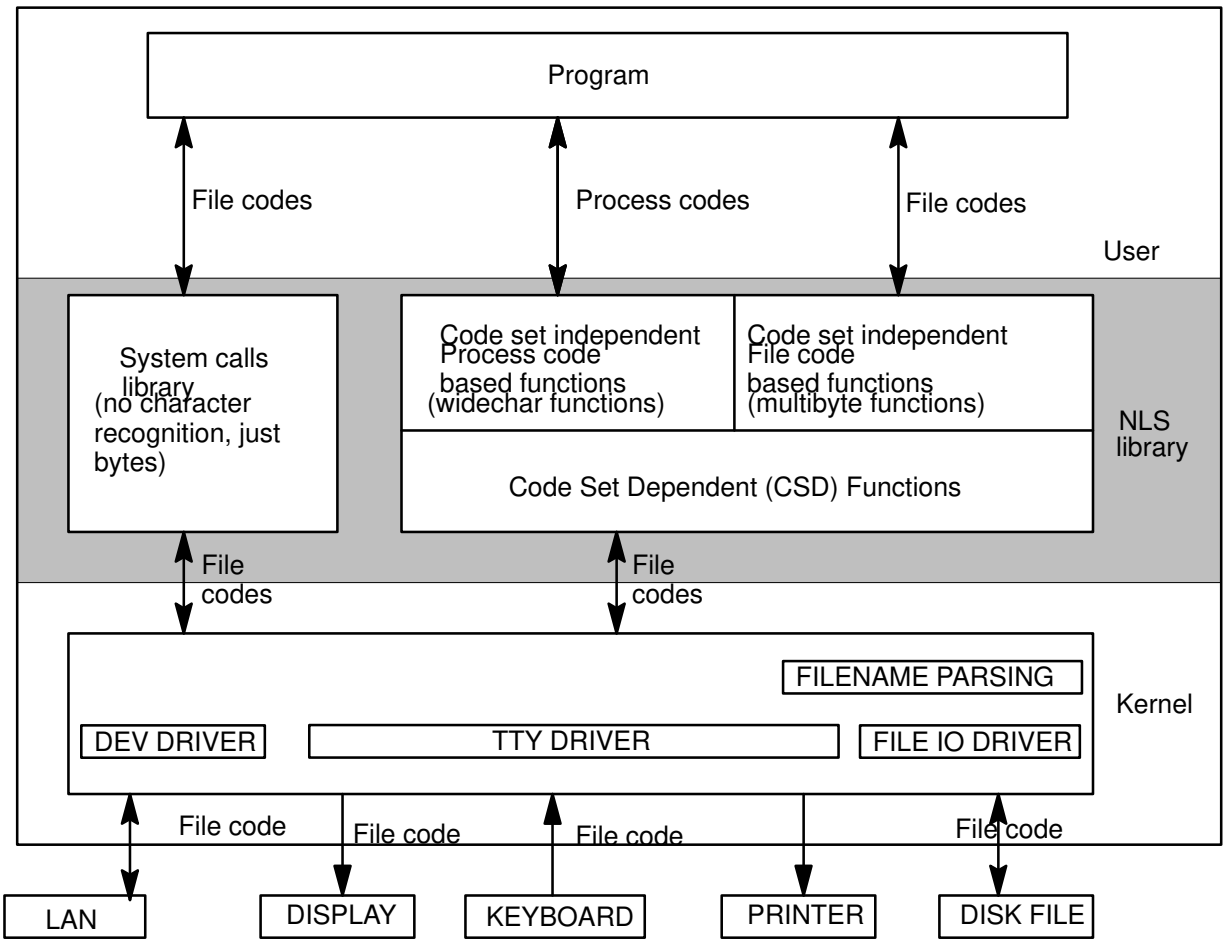


Figure 5. Internationalization Model

4.5 International Subsystems

The following is an illustration of the International Language Support subsystems and the public interfaces that are provided by XPG4 I18N. The model can be compared with Figure 4

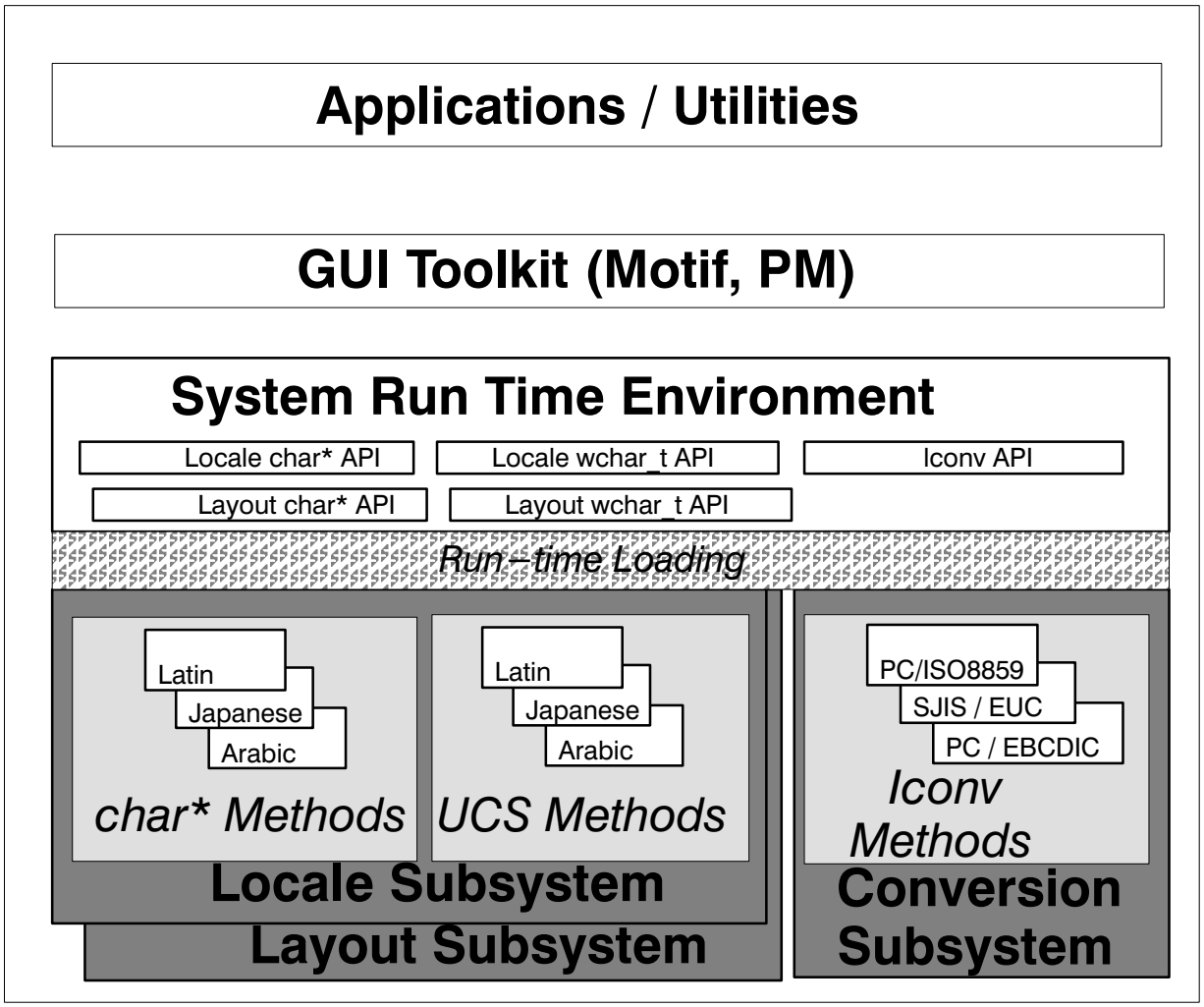


Figure 6. International Subsystems

Section II: ULS Technical Reference

5 Introduction

The **ULS** functions are capable of supporting applications which deal with multiple processing contexts (e.g. locales), simultaneously, to address **multilingual**, **multithreading**, and **distributed systems**.

Instead of referencing a global processing context, these functions have an additional object handle argument.

5.1 Objectives

- Satisfy multilingual and multithreading processing.
- Satisfy the multi−locale support requirements encountered in the windowing environment.
- Coexist with the ”global locale” functions.

5.2 Non−objectives

- The provision of a data tagging mechanism is not an objective.

5.3 Assumptions

- Only Unicode is handled for processing.
- All character data are specified, processed, and otherwise manipulated as **UniChar**.

6 Locale Functions

6.1 Locale Management Functions

The locale management functions manage the connection between the external definitions of locales and the functions that use those definitions.

The **UniCreateLocaleObject()** function allocates system resources associated with the specified locale and provides the caller with a locale object handle. This locale object handle is used by other functions to control their locale behavior for the duration of the function call. An application is allowed to create multiple locale object handles. Each locale object handle may refer to a different locale, thus allowing the application to take advantage of multiple and concurrent locale support.

The **UniQueryLocaleObject()** function obtains information about the locale categories of a particular locale object.

The **UniFreeLocaleObject()** function deallocates resources associated with locale objects.

The **UniLocaleStrToToken()** function converts a locale name string into a locale token value.

The **UniLocaleTokenToStr()** function converts a locale token value to a locale name string.

6.1.1 UniCreateLocaleObject() – Create a locale object

```
#include <UniLoc.h>
int UniCreateLocaleObject(
        int LocaleSpecType,
        const void *LocaleSpec,
        LocaleObject *locale_object
        );
```

The **UniCreateLocaleObject()** function creates a locale object for the locale category or categories specified by **LocaleSpec**. The object created is an opaque object containing all the data and methods necessary to perform the language sensitive operations of functions which accept an argument of type **LocaleObject**. If the function is successful, all categories of the locale object are created and initialized.

The **LocaleSpec** argument points to either a character string or a locale token, as indicated by the value of the **LocaleSpecType** argument. The value of the **LocaleSpec** argument may be obtained by calling one of the functions **UniLocaleStrToToken()** or **UniLocaleTokenToStr()**.

The constant names of the values of **LocaleSpecType** are defined in the header **UniLoc.h**:

UNI_TOKEN_POINTER	**LocaleSpec** points to a locale token.
UNI_MBS_STRING_POINTER	**LocaleSpec** points to a multibyte character string.
UNI_UCS_STRING_POINTER	**LocaleSpec** points to a UCS character string.

When the **LocaleSpec** argument is a pointer to a character string (UCS character string or multibyte character string) it identifies the name of the locale to be initialized. The locale name is used to locate physical resources associated with this locale. The locale name **UNIVERSAL** is reserved and refers to the definitions which provide default behavior for functions.

When the **LocaleSpecType** argument indicates a string and the **LocaleSpec** argument points to a null string, the **UniCreateLocaleObject()** function searches the process environment to resolve the locale object category values according to the following precedence:

- If the **LC_ALL** environment variable is defined and is not null, the value of the LC_ALL environment variable is used.

- If the **LC_*** environment variable(s) (**LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_MONETARY**, **LC_NUMERIC**, **LC_MESSAGES**) are defined and not null, the value of the environment variable is used to initialize the category that corresponds to the environment variable.

- If the **LANG** environment variable is defined and is not null, the value of the **LANG** environment variable is used.

The **LANG** environment variable has the lowest precedence of all of the environment variables used by the **UniCreateLocaleObject()** function to initialize locale categories.

In the absence of the **LC_ALL** environment variable, the value of the **LANG** environment variable is used to determine the value of the category for which the environment variable (**LC_CTYPE**, **LC_COLLATE**, **LC_TIME**, **LC_MONETARY**, **LC_NUMERIC**, **LC_MESSAGES**) that corresponds to that category is not defined or is null.

When the **LocaleSpec** argument is a NULL pointer (without regard to the value of the **LocaleSpecType** argument), the **UniCreateLocaleObject()** function creates a locale object for the **UNIVERSAL** locale.

When the **LocaleSpec** argument points to a locale token value as indicated by the value of the **LocaleSpecType** argument, the token identifies the locale to be initialized.

If the specified locale is valid and supported, the **UniCreateLocaleObject()** function allocates memory for the new object and returns the address of the created locale object in the area pointed to by **locale_object**. It is the application's responsibility to free this memory with a call to the **UniFreeLocaleObject()** function when the object is no longer needed. If the function fails for any reason, the contents of the area pointed to by **locale_object** are undefined.

The **UniCreateLocaleObject()** function returns one of the following values:

UNI_SUCCESS	the specified locale is supported and a valid locale object was created.
UNI_UNSUPPORTED	the specified locale is not supported; the locale object pointer points to undefined data.
UNI_NOMEMORY	there is insufficient memory to create the requested locale or the default locale; the locale object pointer points to undefined data.
UNI_INVALID	an invalid locale specification string or token was passed; the locale object pointer points to undefined data.

6.1.2 UniQueryLocaleObject() – Retrieve the locale name

```
#include <UniLoc.h>
int UniQueryLocaleObject(
        const LocaleObject locale_object,
        int category,
        int LocaleSpecType,
        void **locale_name
        );
```

The **UniQueryLocaleObject()** function returns a pointer to a locale specification in the area pointed to by **locale_name**. The **UniQueryLocaleObject()** function allocates memory to hold the generated value as necessary. It is the application's responsibility to free the memory with the **free()** function.

The value returned in the area pointed to by **locale_name** will point to either a string or a token, as indicated by the value of the **LocaleSpecType** argument.

The **LocaleSpecType** argument can take any of the following values, which are constants defined in the header **UniLoc.h**:

UNI_TOKEN_POINTER	requests that a pointer to a token pointer be returned.
UNI_MBS_STRING_POINTER	requests that a multibyte string pointer be returned.
UNI_UCS_STRING_POINTER	requests that a UCS string pointer be returned.

The permissible values for **category** are: **LC_ALL**, **LC_COLLATE, LC_CTYPE, LC_MESSAGES, LC_MONETARY, LC_NUMERIC,** and **LC_TIME.**

When the **LocaleSpecType** argument is UNI_TOKEN_POINTER and the **category** argument is valid, a pointer to a token which represents the locale value associated with the **category** argument is returned if such a token exists.

When the **LocaleSpecType** argument is UNI_MBS_STRING_POINTER or UNI_UCS_STRING_POINTER, the **UniQueryLocaleObject()** function returns a pointer to a string which represents the locale value associated with the **category** argument.

When the **LocaleSpecType** argument is UNI_MBS_STRING_POINTER or UNI_UCS_STRING_POINTER and the **category** argument is **LC_ALL** a string which represents the values of all of the locale categories of **locale_object** is returned. The returned string may be used as the **LocaleSpec** argument to the **UniCreateLocaleObject()** function to cause the creation of a locale object which is a functional equivalent of **locale_object**.

When the **LocaleSpecType** argument is UNI_MBS_STRING_POINTER or UNI_UCS_STRING_POINTER and the **category** argument is **LC_COLLATE**, **LC_CTYPE**, **LC_MESSAGES**, **LC_MONETARY**, **LC_NUMERIC**, or **LC_TIME**, a string which represents the value of the respective locale category of **locale_object** is returned. The returned string may be used as the **LocaleSpec** argument to the **UniCreateLocaleObject()** function to cause the creation of a locale object with all locale category values set to the value of the queried locale category of **locale_object.**

If **locale_object** contains a null pointer, the **UniQueryLocaleObject()** function returns a locale specification pointer identifying the respective category or categories of the the global locale. If the category argument is **LC_ALL**, this value can be passed to the **UniCreateLocaleObject()** function to create a locale object that is the functional equivalent of the current global locale.

If **locale_object** is invalid, the contents of **locale_name** are undefined and no memory is allocated.

The **UniQueryLocaleObject()** function returns one of the following values:

UNI_SUCCESS	a valid locale specification for the supplied locale object is returned.
UNI_NOMEMORY	insufficient memory to store the locale specification.
UNI_BADOBJ	invalid locale object specified.
UNI_NOTOKEN	no token can be assembled for the locale object and category combination. The contents of **locale_name** are undefined and no memory is allocated.
UNI_UNSUPPORTED	the specified category is not supported.

6.1.3 UniFreeLocaleObject() – Free a locale object

```
#include <UniLoc.h>
int UniFreeLocaleObject(
        LocaleObject locale_object
        );
```

The **UniFreeLocaleObject()** function destroys the locale object identified by **locale_object** and frees any memory associated with it.

The **UniFreeLocaleObject()** function returns one of the following values:

UNI_SUCCESS	the specified locale object has been successfully destroyed.
UNI_BADOBJ	the locale object specified by **locale_object** is not a valid locale object.

6.1.4 UniLocaleStrToToken() – Locale specification string to token

```
#include <UniLoc.h>
int UniLocaleStrToToken(
        int LocaleStringType,
        const void *locale_string,
        LocaleToken *locale_token
        );
```

The **UniLocaleStrToToken()** function accepts as an argument a locale string qualified by the value of the **LocaleStringType** argument. It returns a locale token pointed to by **locale_token** if such a token exists for that locale string. The **UniLocaleStrToToken()** function allocates memory to hold the locale token value. It is the application's responsibility to free the memory with the **free()** function when the locale token value is no longer needed. If no locale token exists for the supplied locale string, the value returned in **locale_token** is undefined.

The **LocaleStringType** argument can take any of the following values, which are constants defined in the header **UniLoc.h**:

UNI_MBS_STRING_POINTER	requests that a multibyte string pointer be returned.
UNI_UCS_STRING_POINTER	requests that a UCS string pointer be returned.

The **UniLocaleStrToToken()** function returns one of the following values:

UNI_SUCCESS	the locale string was valid and an appropriate locale token was returned.
UNI_NOTOKEN	no appropriate locale token exists for this locale string.
UNI_NOMEMORY	insufficient memory to store the locale token.

6.1.5 UniLocaleTokenToStr() – Locale token to locale string

```
#include <UniLoc.h>
int UniLocaleTokenToStr(
        const LocaleToken locale_token,
        int LocaleStringType,
        void **locale_string,
        );
```

The **UniLocaleTokenToStr()** function accepts as an argument a locale token in **locale_token** and returns a pointer to a locale string in **locale_string** qualified by the **LocaleStringType** argument. The **UniLocaleTokenToStr()** function allocates memory to hold the locale string value. It is the application's responsibility to free the memory with the **free()** function when the locale string value is no longer needed. If no locale string can be generated for the supplied locale token, the value returned in **locale_string** is undefined.

The **LocaleStringType** argument can take any of the following values, which are constants defined in the header **UniLoc.h**:

UNI_MBS_STRING_POINTER	requests that a multibyte string pointer be returned.
UNI_UCS_STRING_POINTER	requests that a UCS string pointer be returned.

The **UniLocaleTokenToStr(** function returns one of the following values:

UNI_SUCCESS	a locale string for the locale token was successfully found and returned.
UNI_INVALID	the locale token supplied could not be matched to a locale string.
UNI_NOMEMORY	insufficient memory to store the locale string.

6.1.6 Locale Token Format

The locale token provides a shorthand notation for specifying a locale. The format of the locale token is as defined in the **X/Open Distributed Internationalization Services** (DISS): an unsigned integer of four octets.

6.2 Locale Information Functions

6.2.1 UniQueryLocaleInfo() − Retrieve locale conventions information

```
#include <UniChar.h>
int UniQueryLocaleInfo(
        const LocaleObject locale_object,
        struct UniLconv **UniLconv_struct
        );
```

The **UniQueryLocaleInfo()** function retrieves information from the locale indicated by the **locale_object** argument and places the information in an **UniLconv** structure. The **UniQueryLocaleInfo()** function allocates memory to hold the **UniLconv** structure. It is the application's responsibility to free the memory with the **UniFreeLocaleInfo()** function when the **UniLconv** structure is no longer needed. The address of the **UniLconv** structure is returned in **UniLconv_struct**. The **UniLconv** structure is filled in according to the locale indicated by the locale object handle argument.

The **UniLconv** structure contains the following members:

```
UniChar *decimal_point;          /* non−monetary decimal point */
UniChar *thousands_sep;          /* non−monetary thousands separator */
UniChar *grouping;               /* non−monetary size of grouping */
UniChar *int_curr_symbol;        /* international currency sym and separator */
UniChar *currency_symbol;        /* local currency symbol */
UniChar *mon_decimal_point;      /* monetary decimal point */
UniChar *mon_thousands_sep;      /* monetary thousands separator */
UniChar *mon_grouping;           /* monetary size of grouping */
UniChar *positive_sign;          /* non−negative values sign */
UniChar *negative_sign;          /* negative values sign */
UniChar int_frac_digits;         /* number of fractional digits − int currency */
UniChar frac_digits;             /* number of fractional digits − local currency */
UniChar p_cs_precedes;           /* (non−neg curr sym) 1−precedes, 0−succeeds */
UniChar p_sep_by_space;          /* (non−neg curr sym) 1−space, 0−no space */
UniChar n_cs_precedes;           /* (neg curr sym) 1−precedes, 0−succeeds */
UniChar n_sep_by_space;          /* (neg curr sym) 1−space, 0−no space */
UniChar p_sign_posn;             /* positioning of non−negative monetary sign */
UniChar n_sign_posn;             /* positioning of negative monetary sign */
UniChar *debit_sign;             /* nonnegative−valued monetary symbol − "DB" */
UniChar *credit_sign;            /* negative−valued monetary symbol − "CR" */
UniChar *left_parenthesis;       /* negative−valued monetary symbol − "(" */
UniChar *right_parenthesis;      /* negative−valued monetary symbol − ")" */
```

The value of grouping and mon_grouping is interpreted according to the following:

0xffff No further grouping is to be performed.

0x0000 The previous element is to be repeatedly used for the remainder of the digits.

other The integer value is the number of digits that comprise the current group. The next element is examined to determine the size of the next group of digits before the current group.

The n_sign_posn and p_sign_posn elements are interpreted according to the following:

0 – the quantity and currency_symbol are enclosed in parentheses

1 – the sign precedes the quantity and currency_symbol

2 – the sign follows the quantity and currency_symbol

3 – the sign precedes the currency_symbol

4 – the sign follows the currency_symbol

5 – use debit or credit sign for p_sign_posn or n_sign_posn

The **UniQueryLocaleInfo()** function returns one of the following values:

UNI_SUCCESS	the **UniLconv** structure successfully filled with items associated with the locale object **locale_object**.
UNI_BADOBJ	the locale object specified by **locale_object** is not a valid locale object.
UNI_NOMEMORY	there is insufficient memory to create the **UniLconv** structure; the **UniLconv_struct** pointer points to undefined data.

6.2.2 UniFreeLocaleInfo() – Free UniLconv structure

```
#include <UniChar.h>
int UniFreeLocaleInfo(
        struct UniLconv *UniLconv_struct
        );
```

The **UniFreeLocaleInfo()** function frees the **UniLconv** structure and all associated memory.

The **UniFreeLocaleInfo()** function returns one of the following values:

UNI_SUCCESS	the **UniLconv** structure and associated memory successfully released.
UNI_BADOBJ	the object specified by **UniLconv_struct** is not a valid **UniLconv** structure.

6.2.3 UniQueryLocaleItem() – Retrieve locale information by item

```
#include <nl_types.h>
#include <LangInfo.h>
#include <UniChar.h>
int UniQueryLocaleItem(
        const LocaleObject locale_object,
        nl_item item,
        UniChar **info_item
        );
```

The **UniQueryLocaleItem()** function returns a pointer in **info_item** to a null–terminated **UniChar** string containing information found in the locale object identified by **locale_object** about the language or cultural item named by the **item** argument. The **UniQueryLocaleItem()** function allocates the memory to hold the **UniChar** string and returns a pointer in **info_item**. It is the application's responsibility to free the memory with the **free()** function when the **UniChar** string is no longer needed.

The constant names and values for **item** are contained in **LangInfo.h**:

D_T_FMT	– string for formatting date and time
D_FMT	– string for formatting date
T_FMT	– string for formatting time
AM_STR	– string for a.m.
PM_STR	– string for p.m.
DAY_1	– name of the first day of the week (Sunday)
DAY_2	– name of the second day of the week (Monday)
DAY_3	– name of the third day of the week (Tuesday)
DAY_4	– name of the fourth day of the week (Wednesday)
DAY_5	– name of the fifth day of the week (Thursday)
DAY_6	– name of the sixth day of the week (Friday)
DAY_7	– name of the seventh day of the week (Saturday)
ABDAY_1	– abbreviated first day of the week (Sun)
ABDAY_2	– abbreviated second day of the week (Mon)
ABDAY_3	– abbreviated third day of the week (Tue)
ABDAY_4	– abbreviated fourth day of the week (Wed)
ABDAY_5	– abbreviated fifth day of the week (Thu)
ABDAY_6	– abbreviated sixth day of the week (Fri)
ABDAY_7	– abbreviated seventh day of the week (Sat)
MON_1	– name of the first month (January)
MON_2	– name of the second month (February)
MON_3	– name of the third month (March)
MON_4	– name of the fourth month (April)
MON_5	– name of the fifth month (May)
MON_6	– name of the sixth month (June)

MON_7	– name of the seventh month (July)
MON_8	– name of the eighth month (August)
MON_9	– name of the ninth month (September)
MON_10	– name of the tenth month (October)
MON_11	– name of the eleventh month (November)
MON_12	– name of the twelfth month (December)
MON_13	– name of the thirteenth month
ABMON_1	– abbreviated first month (Jan)
ABMON_2	– abbreviated second month (Feb)
ABMON_3	– abbreviated third month (Mar)
ABMON_4	– abbreviated fourth month (Apr)
ABMON_5	– abbreviated fifth month (May)
ABMON_6	– abbreviated sixth month (Jun)
ABMON_7	– abbreviated seventh month (Jul)
ABMON_8	– abbreviated eighth month (Aug)
ABMON_9	– abbreviated ninth month (Sep)
ABMON_10	– abbreviated tenth month (Oct)
ABMON_11	– abbreviated eleventh month (Nov)
ABMON_12	– abbreviated twelfth month (Dec)
ABMON_13	– abbreviated thirteenth month
RADIXCHAR	– radix character
THOUSEP	– separator for thousands
YESSTR	– affirmative response for yes/no queries
NOSTR	– negative response for yes/no queries
CRNCYSTR	– currency symbol; – leading, + trailing
CODESET	– current code set of the process

The **UniQueryLocaleItem()** function returns one of the following values:

UNI_SUCCESS	the **UniChar** string is successfully filled with the requested item associated with the locale object **locale_object**.
UNI_BADOBJ	the locale object specified by **locale_object** is not a valid locale object.
UNI_UNSUPPORTED	the item value specified by the **item** argument is not supported.
UNI_NOMEMORY	there is insufficient memory to store the **UniChar** string; the **info_item** pointer points to undefined data.

6.3 Date and Time

6.3.1 UniStrftime() − Format date and time

```
#include <time.h>
#include <UniChar.h>
size_t UniStrftime(
            const LocaleObject locale_object,
            UniChar *ucs,
            size_t maxsize,
            const  UniChar *format,
            const struct tm *timeptr
            );
```

Convert the internal time and date specification into a character string and place the results in the area pointed to by **ucs** under the direction of **format**. The null−terminated result of at most **maxsize** code elements is placed in the memory location addressed by **ucs**. If the total number of resulting code elements including the null code element is not more than **maxsize**, the number of code elements placed into the memory location addressed by **ucs** not including the null code element is returned. Otherwise, zero is returned and the contents of the memory location are indeterminate. The format string may contain conversion specification characters and characters that are placed unchanged into the array. The characters that are converted are determined by the **LC_TIME** category of the locale indicated by the locale object handle argument **locale_object** and by the values in the time structure pointed to by **timeptr**. The behavior is undefined when objects being copied overlap. The conversion specifications and their meanings are those described in ANSI−C and XPG4V2 for the **strftime()** function. If data has the form of a directive, but is not one of those described in ANSI−C or XPG4V2 for the **strftime()** function, the character(s) following the **%** is copied to the output.

6.3.2 UniStrptime() − Date and time conversion

```
#include <UniChar.h>
#include <time.h>
 UniChar *UniStrptime(
            const LocaleObject locale_object,
            const  UniChar *buf,
            const  UniChar *fmt,
            struct tm *tm
            );
```

The **UniStrptime()** function converts the character string pointed to by **buf** to a time value, which is stored in the structure pointed to by **tm**, using the format specified by **fmt**. A pointer to the character following the last character in the string pointed to by **buf** is returned. The character string pointed to by **fmt** consists of field descriptors and text characters, reminiscent of the **scanf()** function. Each field descriptor consists of a **%** character followed by another character that specifies the replacement for the field descriptor. Upon successful completion **UniStrptime()** returns a pointer to the character following the last character; otherwise, a null pointer is returned. The supported field descriptors are the same as those described in XPG4V2 for the **strptime()** function. Case is ignored when matching items in **buf** such as month or weekday names. If the conversion fails, contents of the structure pointed to by **tm** is undefined.

6.4 Monetary Formatting

6.4.1 UniStrfmon() – Convert monetary value to string

```
#include <monetary.h>
#include <UniChar.h>
int UniStrfmon(
        const LocaleObject locale_object,
        UniChar *ucs,
        size_t maxsize,
        const  UniChar *format,
        ...
        );
```

The **UniStrfmon()** function places characters into the array pointed to by **ucs** as controlled by the string pointed to by **format**. No more than **maxsize** code elements are placed into the array. The character string format contains two types of objects: plain characters, which are copied to the output stream, and directives, 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 ignored. A directive consists of a **%** character, optional conversion specifications, and a terminating character that determines the directive's behavior. Each directive is replaced by the appropriate characters as described for the **strfmon()** function in XPG4V2.

7 String/Character Functions

7.1 String Manipulation Functions

The string manipulation functions provide for the handling of strings of text (i.e., arrays of code elements of type **UniChar**). Array length is always determined in terms of the number of code elements in the array, that is, the number of code elements up to the null code element. A null code element is a code element of value 0x0000. A UniChar * or void * argument always points to the initial (lowest addressed) code element of a **UniChar** array. If an array is written beyond the end of an object the behavior is undefined.

7.1.1 UniStrcpy() – Copy code element string

```
#include <UniChar.h>
UniChar *UniStrcpy(
        UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrcpy()** function copies the code element string pointed to by **ucs2** (including the terminating null code element) into the code element array pointed to by **ucs1**. If copying takes place between objects that overlap, the behavior is undefined.

The **UniStrcpy()** function returns the value of **ucs1**.

7.1.2 UniStrncpy() – Copy a specific number of code elements

```
#include <UniChar.h>
UniChar *UniStrncpy(
        UniChar *ucs1,
        const UniChar *ucs2,
        size_t n
        );
```

The **UniStrncpy()** function copies not more than **n** code elements (code elements that follow a null code element are not copied) from the code element array pointed to by **ucs2** to the code element array pointed to by **ucs1**. If copying takes place between objects that overlap, the behavior is undefined. If the code element array pointed to by **ucs2** is a code element string that is shorter than **n** code elements, null code elements are appended to the copy in the code element array pointed to by **ucs1**, until **n** code elements in all have been written.

The **UniStrncpy()** function returns the value of **ucs1**.

7.1.3 UniStrcat() – Concatenate code element strings

```
#include <UniChar.h>
UniChar *UniStrcat(
        UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrcat()** function appends a copy of the code element string pointed to by **ucs2** (including the terminating null code element) to the end of the code element string pointed to by **ucs1**. The initial wide code element of **ucs2** overwrites the null code element at the end of **ucs1**. If copying takes place between objects that overlap, the behavior is undefined.

The **UniStrcat()** function returns the value of **ucs1**.

7.1.4 UniStrncat() – Concatenate a specific number of code elements

```
#include <UniChar.h>
UniChar *UniStrncat(
        UniChar *ucs1,
        const UniChar *ucs2,
        size_t n
        );
```

The **UniStrncat()** function appends not more than **n** code elements (a null code element and code elements that follow it are not appended) from the code element array pointed to by **ucs2** to the end of the code element string pointed to by **ucs1**. The initial code element of **ucs2** overwrites the null code element at the end of **ucs1**. A terminating null code element is always appended to the result. If copying takes place between objects that overlap, the behavior is undefined.

The **UniStrncat()** function returns the value of **ucs1**.

7.1.5 UniStrcmp() – Compare code element strings

```
#include <UniChar.h>
int UniStrcmp(
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrcmp()** function compares the code element string pointed to by **ucs1** to the code element string pointed to by **ucs2**.

The **UniStrcmp**() function returns an integer greater than, equal to, or less than zero, according to whether the code element string pointed to by **ucs1** is greater than, equal to, or less than the code element string pointed to by **ucs2**.

7.1.6 UniStrncmp() – Compare a specific number of code elements

```
#include <UniChar.h>
int UniStrncmp(
        const UniChar *ucs1,
        const UniChar *ucs2,
        size_t n
        );
```

The **UniStrncmp()** function compares not more than **n** code elements (code elements that follow a null code element are not compared) from the code element array pointed to by **ucs1** to the code element array pointed to by **ucs2**.

The **UniStrncmp()** function returns an integer greater than, equal to, or less than zero, according to whether the possibly null–terminated code element array pointed to by **ucs1** is greater than, equal to, or less than the possibly null–terminated code element array pointed to by **ucs2**.

7.2 String Search Functions

7.2.1 UniStrchr() – Search for the first occurrence of a code element.

```
#include <UniChar.h>
UniChar *UniStrchr(
        const UniChar *ucs,
        UniChar uc
        );
```

The **UniStrchr()** function locates the first occurrence of **uc** in the array of code elements pointed to by **ucs**. The terminating null code element is considered to be part of the string.

The **UniStrchr()** function returns a pointer to the located code element, or a null pointer if the code element does not occur in the string.

7.2.2 UniStrcspn() – Search for code element string

```
#include <UniChar.h>
size_t UniStrcspn(
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrcspn()** function computes the length of the maximum initial segment of the code element string pointed to by **ucs1** which consists entirely of code elements not from the string pointed to by **ucs2**.

The **UniStrcspn()** function returns the length of the segment.

7.2.3 UniStrpbrk() – Locate code elements in a code element string

```
#include <UniChar.h>
UniChar *UniStrpbrk(
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrpbrk()** function locates the first occurrence in the string pointed to by **ucs1** of any code element from the code element string pointed to by **ucs2**.

The **UniStrpbrk()** function returns a pointer to the code element, or a null pointer if no code element from the code element string from **ucs2** occurs in **ucs1**.

7.2.4 UniStrrchr() – Locate last occurrence of code element

```
#include <UniChar.h>
UniChar *UniStrrchr(
        const UniChar *ucs,
        UniChar uc
        );
```

The **UniStrrchr()** function locates the last occurrence of **uc** in the code element string pointed to by **ucs**. The terminating null code element is considered to be part of the string.

The **UniStrrchr()** function returns a pointer to the code element, or a null pointer if **uc** does not occur in the code element string.

7.2.5 UniStrspn() – Determine the number of code elements in a segment

```
#include <UniChar.h>
size_t UniStrspn(
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrspn()** function computes the length of the maximum initial segment of the code element string pointed to by **ucs1** which consists entirely of code elements from the code element string pointed to by **ucs2**.

The **UniStrspn()** function returns the length of the segment.

7.2.6 UniStrstr() – Locate a code element sequence

```
#include <UniChar.h>
UniChar *UniStrstr(
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrstr()** function locates the first occurrence in the code element string pointed to by **ucs1** of the sequence of code elements (excluding the terminating null code element) in the code element string pointed to by **ucs2**.

The **UniStrstr()** function returns a pointer to the located code element string, or a null pointer if the string is not found.

If **ucs2** points to a code element string with zero length, the function returns **ucs1**.

7.2.7 UniStrlen() – Determine code element count

```
#include <UniChar.h>
size_t UniStrlen(
        const UniChar *ucs
        )
```

The **UniStrlen()** function computes the length of the code element string pointed to by **ucs**.

The **UniStrlen()** function returns the number of code elements that precede the terminating null code element.

7.3 Composite Character Sequence (CCS) Functions

The functions in this section manipulate composite character sequences. A composite character sequence consists of a non−combining (base) character followed by zero or more combining characters (floating diacritics). The non−combining character and the following combining characters of the composite character sequence are represented by corresponding code elements. Code elements are represented in the **UniChar** data type. A null code element is a code element with all zero bits. This definition is generalized to cover the character code processing model in that in its simplest form a composite character sequence corresponds to a single code element (process code or **UniChar** data type entity) which contains a non−combining (base) character.

7.3.1 UniQueryCompCharLen() − Composite character sequence code element count

```
#include <UniChar.h>
size_t UniQueryCompCharLen(
        const LocaleObject locale_object,
        const UniChar *ucs
        );
```

The **UniQueryCompCharLen()** function computes the number of the code elements in the composite character sequence pointed to by **ucs**.

The **UniQueryCompCharLen()** function returns the number of code elements including the non−combining character and zero or more following combining characters up to, but not including, the next non−combining character or null code element.

If **ucs** is a null pointer or if **ucs** points to a null code element or if **ucs** points to a combining character, the **UniQueryCompCharLen()** function returns 0 .

7.3.2 UniNextCompChar() − Advance to next composite character sequence

```
#include <UniChar.h>
UniChar *UniNextCompChar(
        const LocaleObject locale_object,
        const UniChar *ucs
        );
```

The **UniNextCompChar()** function locates the next non−combining character in a string.

If **ucs** is not a null pointer, the **UniNextCompChar()** function returns a pointer to the next non−combining character in the string pointed to by **ucs** after skipping the composite character sequence at the head of the string.

If **ucs** is a null pointer or if **ucs** points to a null code element or if **ucs** points to a combining character, the **UniNextCompChar()** function returns a null pointer.

7.3.3 UniQueryCompChar() – Number of composite character sequences

```
#include <UniChar.h>
size_t UniQueryCompChar(
        const LocaleObject locale_object,
        const UniChar *ucs
        );
```

The **UniQueryCompChar()** function computes the number of composite character sequences in the code element array pointed to by **ucs**.

The **UniQueryCompChar()** function returns the number of sequences that consist of a base character followed by zero or more combining characters up to, but not including, the null code element.

If **ucs** is a null pointer or if **ucs** points to a null code element or if **ucs** points to a combining character, the **UniQueryCompChar()** function returns 0.

7.3.4 UniQueryCompStrDWidth() – Display width of a composite character sequence string

```
#include <UniChar.h>
int UniQueryCompStrDWidth(
        const LocaleObject locale_object,
        const UniChar *ucs,
        size_t n
        );
```

The **UniQueryCompStrDWidth()** function determines the number of presentation column positions required for **n** composite character sequences (or fewer composite character sequences if a null code element is encountered before the **n** composite character sequences are exhausted) in the string pointed to by **ucs**.

If **ucs** is not a null pointer, the **UniQueryCompStrDWidth()** function either returns 0 (if **ucs** points to a null code element or if **ucs** points to a combining character), or returns the number of display columns to be occupied by the **n** or fewer composite character sequences of the string pointed to by **ucs**.

If **ucs** is a null pointer the **UniQueryCompStrDWidth()** function returns 0.

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.

7.4 Character Attribute Functions

The functions described in this section provide a generalized character attribute query.

The **UniQueryCharAttr()** function allows a generalized query of the attributes of all characters. The attribute value of a given character is returned.

7.4.1 UniCreateAttrObject() – Initialize for attribute query

```
#include <UniChar.h>
int UniCreateAttrObject(
        const LocaleObject locale_object,
        const UniChar *AttrName,
        AttrObject *attr_object
        );
```

The **UniCreateAttrObject()** function allocates resources associated with an attribute defined in the **LC_CTYPE** category of the locale indicated by the **locale_object** argument.

The **locale_object** argument specifies a locale object handle returned by the **UniCreateLocaleObject()** function. It should not be a NULL pointer.

The **AttrName** argument specifies the attribute name(s) for which an attribute object handle should be created. Multiple attribute names are specified as a string of space separated names.

When the **UniCreateAttrObject()** function completes without errors, the **attr_object** argument specifies a valid pointer to an attribute object.

The attribute object pointer should be used in all subsequent calls to the **UniQueryCharAttr()** function. If the function result is other than UNI_SUCCESS, the attribute object pointer specifies a NULL pointer.

The following attribute names shall have their defined meaning in all locales:

alnum	true when **alpha** or **digit** is true
alpha	true when **upper** or **lower** is true, or when none of **cntrl**, **digit**, **punct**, or **space** is true.
blank	true for the characters space and horizontal tab.
cntrl	true for any control character; the following attributes must be false: **upper**, **lower**, **alpha**, **digit**, **xdigit**, **graph**, **print**, **punct.**
digit	true for the digits 0, 1, 2 3, 4, 5, 6, 7, 8, and 9.
graph	true for any character with the **print** attribute, except the space character (code element 0x0020).
lower	true for any character that is a lower−case letter and none of **cntrl**, **digit**, **punct**, or **space** is true.
print	true for **upper**, **lower**, **alpha**, **digit**, **xdigit**, **punct**, or any printing character including the space character (code element 0x0020).
punct	true for any printing character that is neither the space character (code element 0x0020) nor a character for which **alnum** is true.
space	true for any character that corresponds to a standard white space character or is one of the set of white space characters in the locale as indicated by the

	locale_object argument for which **alnum** is false. The standard white space characters are the following: space, form feed, newline, carriage return, horizontal tab and vertical tab.
upper	true for any character that is an upper−case letter and none of **cntrl**, **digit**, **punct**, or **space** is true.
xdigit	true for 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E F, a, b, c, d, e, and f.

In addition to the above attribute names, other attribute names in the locale (including user defined attributes) may be passed to the **UniCreateAttrObject()** function via the **AttrName** argument. In order to obtain a successful return, the attribute name must be defined in **locale_object**.

The **UniCreateAttrObject()** function returns one of the following:

UNI_SUCCESS	There are no errors. The **attr_object** argument points to a valid attribute object.
UNI_UNSUPPORTED	the attribute name specified by the **AttrName** argument is not supported in **locale_object**.
UNI_NOMEMORY	there is insufficient memory to create the attribute object; the **attr_object** pointer points to undefined data.

7.4.2 UniQueryCharAttr() – Query character attribute

```
#include <UniChar.h>
int UniQueryCharAttr(
        AttrObject attr_object,
        UniChar uc
    );
```

The **UniQueryCharAttr()** function determines whether the code element **uc** has the attribute(s) specified by the attribute object argument, **attr_object**.

The **attr_object** argument specifies a character attribute object.

The **uc** argument specifies a code element whose attribute should be tested.

If the result of the test is true (the code element has the attribute associated with the attribute object, **attr_object**), the **UniQueryCharAttr()** function returns the value 1.

If the result of the test is false, the **UniQueryCharAttr()** function returns the value 0.

7.4.3 UniScanForAttr() – Scan for attribute match

```
#include <UniChar.h>
int UniScanForAttr(
        AttrObject attr_object,
        const UniChar *ucs,
        size_t num_elems,
        Boolean inverse_op,
        size_t *offset
        );
```

The **UniScanForAttr()** function scans the array of code elements identified by **ucs**, from the position specified by **ucs**, searching for the first code element that matches or does not match the set of attributes specified by **attr_object**.

The **inverse_op** argument 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_op** is set to **False**, the function searches for the first code element that matches all of the attributes of the specified **attr_object**; if **inverse_op** is set to **True**, the function searches for the first code element that matches none of the attributes of **attr_object**.

The search begins from the code element identified by **ucs**, through the next **num_elems** code elements. A non–negative integer identifying the location of the first code element meeting all of the criteria specified by **attr_object** is returned in the area pointed to by **offset**. This indicates the number of code elements offset from the code element identified by **ucs**, to the code element at which the attribute match is satisfied. If no code element meets the specified criteria, the contents of **offset** are undefined.

In addition to the general character attributes described under **UniCreateAttrObject()**, the following attributes which require contextual analysis can be handled by the **UniScanForAttr()** function:

csid	the next code element that has a Charsetid value different than the Charsetid of the current code element
hyphenloc	the next code element after which a line discontinuity may reasonably occur
linebreak	a code element that would cause data to be visually presented on a different line
paragraph	the last code element that is part of the current paragraph
sentence	the last code element that is part of the current sentence
word	the last code element that is part of the current word

The **UniScanForAttr()** function returns one of the following:

UNI_SUCCESS	the function call was successful.
UNI_NOMATCH	no code element meets the specified criteria.
UNI_BADOBJ	the attribute object specified by **attr_object** is not a valid attribute object.

7.4.4 UniFreeAttrObject() – Release character attribute object

#include <UniChar.h>
int UniFreeAttrObject(
AttrObject attr_object
);

The **UniFreeAttrObject()** function releases all resources associated with the character attribute object allocated by the **UniCreateAttrObject()** function.

The **attr_object** argument specifies a previously allocated attribute object.

The **UniFreeAttrObject()** function returns one of the following values:

UNI_SUCCESS	all resources associated with the attribute object specified by the **attr_object** argument have been successfully de−allocated.
UNI_BADOBJ	the attribute object specified by **attr_object** is not a valid attribute object.

7.4.5 Atomic Query Attributes

The following functions provide the functionality of **UniCreateAttrObject()**, **UniQueryCharAttr()**, and **UniFreeAttrObject()** as an atomic operation for the invariant attributes:

int UniQueryAlnum(const LocaleObject locale_object, UniChar uc);

int UniQueryAlpha(const LocaleObject locale_object, UniChar uc);

int UniQueryBlank(const LocaleObject locale_object, UniChar uc);

int UniQueryCntrl(const LocaleObject locale_object, UniChar uc);

int UniQueryDigit(const LocaleObject locale_object, UniChar uc);

int UniQueryGraph(const LocaleObject locale_object, UniChar uc);

int UniQueryLower(const LocaleObject locale_object, UniChar uc);

int UniQueryPrint(const LocaleObject locale_object, UniChar uc);

int UniQueryPunct(const LocaleObject locale_object, UniChar uc);

int UniQuerySpace(const LocaleObject locale_object, UniChar uc);

int UniQueryUpper(const LocaleObject locale_object, UniChar uc);

int UniQueryXdigit(const LocaleObject locale_object, UniChar uc);

7.5 String Transformation Functions

The string transformation functions perform string transformation according to a transformation type defined in the **LC_CTYPE** category of the locale. The external transformation type definitions are associated with those functions using locale object handle arguments. The **upper**, **lower**, **level1**, **level2**, and **level3** transformation types are available in all locales. The **LC_CTYPE** category may also contain definitions of additional transformation types.

7.5.1 UniCreateTransformObject() – Initialize string transformation

```
#include <UniChar.h>
int UniCreateTransformObject(
        const LocaleObject locale_object,
        const  UniChar *xtype,
        XformObject *xform_object
        );
```

The **UniCreateTransformObject()** function obtains a transformation object handle for a transformation type as defined in the locale indicated by the **locale_object** argument. The function returns a transformation object handle which can be used as an argument in the **UniTransformStr()** function.

The l**ocale_object** argument specifies a locale object handle returned by **UniCreateLocaleObject()** function.

The **xtype** argument specifies the transformation type.

The following transformation types shall be defined in all locales:

lower	transform to lowercase characters. A characters which does not have a lowercase form is returned as itself.
upper	transform to uppercase characters. A characters which does not have a uppercase form is returned as itself.
level1	transform level 3 data to composed character (level 1) representation. All composite character sequences which can not be combined into a precomposed form are returned as themselves.
level2	transform level 3 data to level 2 form. All composite character sequences which can not be combined into a form appropriate for level 2 are returned as themselves.
level3	transform the level 1 or level 2 data to base character and floating diacritics.

In addition to the above transformation type names, other transformation type names in the locale (including user defined transformation type names) may be passed to the **UniCreateTransformObject()** function via the **xtype** argument. In order to obtain a successful return, the transformation type name must be defined in **locale_object**.

When the **UniCreateTransformObject()** function completes without errors, the **xform_object** argument value specifies a valid pointer to a transformation object. The transformation object should be used in all subsequent calls to the **UniTransformStr()** function. If the function result is other than UNI_SUCCESS, the transformation object handle is set to a NULL pointer.

The **UniCreateTransformObject()** function returns one of the following:

UNI_SUCCESS	no errors, the **xform_object** argument points to a valid transformation object.
UNI_UNSUPPORTED	the transformation name type specified by the **xtype** argument is not supported for **locale_object**.
UNI_NOMEMORY	there is insufficient memory to create the transformation object; the **xform_object** pointer points to undefined data.

7.5.2 UniTransformStr() – String transform

```
#include <UniChar.h>
int UniTransformStr(
        XformObject xform_object,
        UniChar *InpBuf,
        int *InpSize,
        UniChar *OutBuf,
        int *OutSize
        );
```

The **UniTransformStr()** function transforms a **UniChar** character string as specified by the transformation object handle **xform_object** for the **LC_CTYPE** category. This category applies to the locale object which was used to create the transformation handle **xform_object** (by the **UniCreateTransformObject()** function). The text from the input buffer is transformed and the result is placed in the output buffer. Any characters not included in the transformation type referenced by **xform_object** are moved to the output buffer unchanged.

The **xform_object** argument specifies a transformation object handle returned by the **UniCreateTransformObject()** function.

The **InpBuf** argument specifies the source string to be processed.

The **InpSize** argument, on input, specifies the number of code elements to be transformed. A value of −1 indicates that the input is delimited by a **UniChar** NULL character (0x0000). On return, the value is modified to the actual number of code elements processed in the source string.

The **OutBuf** argument contains the processed output string.

The **OutSize** argument, on input, specifies the size of the output buffer (number of code elements). On return, the value is modified to the actual number of code elements placed in **OutBuf**.

The **UniTransformStr()** function returns one of the following values:

UNI_SUCCESS	transformation completed without errors.
UNI_BUFFER_FULL	source string is not entirely processed. The output buffer is full.

7.5.3 UniFreeTransformObject() – Free string transform object

```
#include <UniChar.h>
int UniFreeTransformObject(
        XformObject xform_object
        );
```

The **UniFreeTransformObject()** function releases all resources associated with a transformation object handle previously obtained by the **UniCreateTransformObject()** function.

The **xform_object** argument specifies a transformation object handle returned by the **UniCreateTransformObject()** function.

The **UniFreeTransformObject()** function returns one of the following values:

UNI_SUCCESS	specifies that all resources associated with the transformation object specified by the **xform_object** argument have been successfully de−allocated.
UNI_BADOBJ	the transformation object specified by **xform_object** is not a valid transformation object.

7.5.4 Atomic Transformation Functions

The following functions provide the functionality of **UniCreateTransformObject()**, **UniTransformStr()**, and **UniFreeTransformObject()** as atomic operations on a single code element:

UniChar UniTransLower(const LocaleObject locale_object, UniChar uc);

UniChar UniTransUpper(const LocaleObject locale_object, UniChar uc);

7.6 String Conversion Functions

In all cases of the string conversion functions, the portion of the string which represents floating point or integer values may contain national digits instead of or, in addition to European digits.

7.6.1 UniStrtod() − Convert character string to double precision floating point

```
#include <UniChar.h>
int UniStrtod(
        const LocaleObject locale_object,
        const UniChar *nptr,
        UniChar **endptr,
        double *result
        );
```

The **UniStrtod()** function converts the initial portion of the string pointed to by **nptr** to double precision floating point representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as indicated by the **space** attribute), a subject sequence resembling a floating−point constant; and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to a floating−point number, and returns the result in the area pointed to by **result**. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

The **UniStrtod()** function returns one of the following values:

UNI_SUCCESS	the function call was successful.
UNI_BADOBJ	invalid locale object specified.
UNI_INVALID	the **endptr** or **result** argument contains an invalid pointer value. The reliable detection of this error is implementation−dependent.
UNI_RANGE	the conversion resulted in an out of range condition.

7.6.2 UniStrtol() – Convert a character string to a long integer

```
#include <UniChar.h>
int UniStrtol(
        const LocaleObject locale_object,
        const UniChar *nptr,
        UniChar **endptr,
        int base,
        long int *result
        );
```

The **UniStrtol()** function converts the initial portion of the string pointed to by **nptr** to long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as indicated by the **space** attribute), a subject sequence resembling an integer determined by the value of base, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an integer, and returns the result in the area pointed to by **result**. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

The **UniStrtol()** function returns one of the following values:

UNI_SUCCESS	the function call was successful.
UNI_BADOBJ	invalid locale object specified.
UNI_INVALID	the **endptr** or **result** argument contains an invalid pointer value. The reliable detection of this error is implementation–dependent.
UNI_RANGE	the conversion resulted in an out of range condition.

7.6.3 UniStrtoul() – Convert a character string to an unsigned long integer

```
#include <UniChar.h>
unsigned long int UniStrtoul(
        const LocaleObject locale_object,
        const UniChar *nptr,
        UniChar **endptr,
        int base,
        unsigned long int *result
        );
```

The **UniStrtoul()** function converts the initial portion of the string pointed to by **nptr** to unsigned long int representation. First it decomposes the input string into three parts: an initial, possibly empty, sequence of white space characters (as indicated by the **space** attribute), a subject sequence resembling an unsigned integer represented in some radix determined by the value of base, and a final string of one or more unrecognized characters, including the terminating null character of the input string. Then it attempts to convert the subject sequence to an unsigned integer, and returns the result in the area pointed to by **result**. A pointer to the final string is stored in the object pointed to by **endptr**, provided that **endptr** is not a null pointer.

The **UniStrtoul()** function returns one of the following values:

UNI_SUCCESS	the function call was successful.
UNI_BADOBJ	invalid locale object specified.
UNI_INVALID	the **endptr** or **result** argument contains an invalid pointer value. The reliable detection of this error is implementation–dependent.
UNI_RANGE	the conversion resulted in an out of range condition.

7.7 String Comparison Functions

7.7.1 UniStrcoll() – Language collation string comparison

```
#include <UniChar.h>
int UniStrcoll(
        const LocaleObject locale_object,
        const UniChar *ucs1,
        const UniChar *ucs2
        );
```

The **UniStrcoll()** function compares the string pointed to by **ucs1** to the string pointed to by **ucs2**, both interpreted as appropriate to the **LC_COLLATE** category of the locale indicated by the locale object handle argument, l**ocale_object**. The **UniStrcoll()** function returns an integer greater than, equal to, or less than zero, according to whether the character string pointed to by **ucs1** is greater than, equal to, or less than the character string pointed to by **ucs2**, when both character strings are interpreted as appropriate to the **LC_COLLATE** category of the locale indicated by the locale object handle argument, **locale_object**.

7.7.2 UniStrxfrm() – Transform a character string

```
#include <UniChar.h>
size_t UniStrxfrm(
        const LocaleObject locale_object,
        UniChar *ucs1,
        const UniChar *ucs2,
        size_t n
        );
```

The **UniStrxfrm()** function transforms the string pointed to by **ucs2** to values which represent character collating weights and places the resulting string into the array pointed to by **ucs1**. The transformation is such that if the **UniStrcmp()** function is applied to two transformed strings, it returns a value greater than, equal to, or less than zero, corresponding to the result of the **UniStrcoll()** function applied to the same two original strings. No more than **n** elements are placed into the resulting array pointed to by **ucs1**, including the terminating null code element. If **n** is zero, **ucs1** is permitted to be a null pointer. If copying takes place between objects that overlap the behavior is undefined.

The **UniStrxfrm()** function returns the length of the transformed string (not including the terminating null code element). If the value returned is **n** or more, the contents of the array pointed to by **ucs1** are indeterminate. If **ucs1** is a null pointer, **UniStrxfrm()** returns the number of elements required to contain the transformed character string.

The **UniStrxfrm()** function is controlled by the **LC_COLLATE** category of the locale as indicated by the locale object handle argument, **locale_object**.

7.8 Character Set ID Function

The character set identifier (CharsetId) function supports applications that need to determine the CharsetId of characters. Identification of a character's CharsetId is also useful to many applications that need to interchange characters based on the character set definition rather than the code set definition.

7.8.1 UniQueryCharSetId() – Determine the CharsetId of character

```
#include <UniChar.h>
int UniQueryCharSetId(
        const LocaleObject locale_object,
        const UniChar uc
        );
```

The **UniQueryCharSetId()** function returns the CharsetId of the character represented by the character code, **uc**. The character code, **uc**, is assumed to be valid character. No validation of the character is done. The return value is an integer value representing the Charsetid of the character.

The **UniQueryCharSetId()** function uses the **LC_CTYPE** category of the locale object.

8 Layout Functions

8.1 Layout Text Functions

Several locales consist of specific scripts whose proper presentation depends on directional controls and/or the sequence of *context dependent* characters. For example, scripts which are context sensitive will render the same character differently depending on how it joins with its neighbors. This requires the same character to take on several shapes which may be defined by UCS. In addition, text not encoded in UCS may have control characters (sequences) that carry directional and/or contextual information externally from the encoded text.

This section describes the functions that are essential for transforming text (containing context dependent and/or directional controls) between different formats. The specific functions and information(tables) are associated with a locale and are encapsulated in an opaque data type **LayoutObject**. All locales supported by the **setlocale**() function of libC shall be associated with a **LayoutObject** of libi18n**.** Applications will be required to initialize a **LayoutObject** prior to using the LayoutText Functions described in this section.

NOTE: In general, the layout functions should only be used by utilities performing operations that involve transformations to or from the text editing form (presented on a screen, perform font charset specific functions, printer, etc...).

Layout Values within a *LayoutObject*

Each **LayoutObject** defines its own layout values which affect the behavior of the **UniTransformLayout**() function and the **UniEditShapeLayout**() functions.

The primary purpose of the **LayoutObject** is to transform the caller's data from one form to the other and to provide information to the application so that it can perform editing operations per the definition of a writing system. However, in order to really take advantage of any **LayoutObject** an application writer needs to learn about its layout values in order to customize a **LayoutObject** to match the characteristics of the data being processed. In addition, some layout values may be changed at application startup or based on interaction with the user, e.g. orientation.

The section *Layout Values* (see below) describes the standard set of layout values that are common to all **LayoutObjects**. It also describes which layout values may be changed after creation of the **LayoutObject**.

Layout values are described using the **LayoutValues** data type which is a pointer to the **LayoutValueRec** data structure.

```
#include <UniChar.h>
typedef struct  {
        LayoutId name;          /* int       – identifies the id of the layout value */
        LayoutValue value;      /* void*     – Data of layout value item */
        }LayoutValueRec,  *LayoutValues
```

The **name** element defines the layout value to be set and the **value** element contains the data to be set. The **LayoutValue** data type is a C language type which is large enough to contain the following: char*, long, int*, or a pointer to a function. The end of the array is indicated by **name** of value zero (0).

The **LayoutSetValues**() function makes it more convenient to set the two members of the *LayoutValueRec* structure. This function is usually specified in a stylized manner to minimize the probability of making a mistake. Do not use expressions with side effects such as auto−increment or auto−decrement within the first argument to the LayoutSetValue function. The LayoutSetValue function can be implemented as a macro that evaluates the first argument twice. For example:

```
LayoutValueRec values[4];
int i = 0;
```

```
LayoutSetValue( values[i], Orientation, ORIENTATION_RTL);  i++;
LayoutSetValue( values[i], TypeOfText, TEXT_IMPLICIT);  i++;
LayoutSetValue( values[i], BidiType, BIDI_UCS);  i++;
LayoutSetValue( values[i], NULL, NULL);
UniSetLayoutValues( layout, values)

/*
 * The following could also be used to set all values at one time.
 *
LayoutSetValue( values[i], AllTextDescriptor,
        ORIENTATION_RTL | TEXT_IMPLICIT | BIDI_UCS |
        SWAPPING | NUMERAL_NOMINAL | TEXT_SHAPED);  i++;
 */
```

8.1.1 UniCreateLayoutObject() – Initialize a layout context

```
#include <UniChar.h>
int UniCreateLayoutObject (
        int LocaleSpecType,
        const void *LocaleSpec,
        LayoutObject* layout_object
        );
```

The **UniCreateLayoutObject()** function creates a **LayoutObject** associated with the locale specified by **LocaleSpec**. The **LayoutObject** is an opaque object containing all the data and methods necessary to perform the layout operations on context dependent/directional characters associated with the code set of **LocaleSpec**.

The **LocaleSpec** and **LocaleSpecType** arguments which identify the layout object are of the same format as defined for **UniCreateLocaleObject**().

When the **UniCreateLayoutObject**() function completes without errors, the value of the **layout_object** argument points to a valid **LayoutObject** that may be used by other layout functions. The returned **LayoutObject** is initialized to an initial state that defines the behavior of the layout functions. The initial state is locale dependent and is described by the layout values (see 8.1.7 Layout Values) that may be queried using the **UniGetLayoutValues**() function. The layout values of the **LayoutObject** may be changed using the **UniSetLayoutValues**() function and any state maintained by the **LayoutObject** is independent from the current global locale set via the **setlocale**() function.

If the function fails for any reason, the contents of the area pointed to by **layout_object** are undefined and a non−zero value is returned.

The **UniCreateLayoutObject**() function returns one of the following values:

0	No errors, **layout_object** points to a valid handle.
LAYOUT_EINVAL	The locale specified by **LocaleSpec**/**LocaleSpecType** is not available.
LAYOUT_EMFILE	{OPEN_MAX} files descriptors are currently open in the calling process.
LAYOUT_ENOMEM	Insufficient storage space is available.

8.1.2 UniFreeLayoutObject() – Free a LayoutObject

```
#include <UniChar.h>
int UniFreeLayoutObject (
        LayoutObject layout_object     /* LayoutObject Handle */
        );
```

The **UniFreeLayoutObject()** function releases all the resources of a **LayoutObject** obtained by the **UniCreateLayoutObject()** function.

The **layout_object** argument specifies a **LayoutObject** returned by the **UniCreateLayoutObject()** function.

The **UniFreeLayoutObject()** function may returns one of the following values:

0	All resources associated with **layout_object**, are successfully de−allocated.
LAYOUT_EFAULT	Specifies that errors occurred while processing the request.

8.1.3 UniSetLayoutValues() – Set Layout Values of a LayoutObject

```
#include <UniChar.h>
int UniSetLayoutValues (
        LayoutObject layout_object,
        LayoutValues values,
        int *index_returned
        );
```

The **UniSetLayoutValues**() function is used to change the layout values of a **LayoutObject**.

The **layout_object** argument specifies a **LayoutObject** returned by the **UniCreateLayoutObject()** function.

The **values** argument specifies the list of layout values (**LayoutValueRec,** see above) that are to be changed. The values are written into the **LayoutObject** and may affect the behavior of subsequent layout functions. **NOTE:** some layout values do alter internal states maintained by a **LayoutObject**.

When the **UniSetLayoutValues()** function completes without errors all values will have been set in the **LayoutObject** and a zero is returned.

If any value can not be set the index in **values** causing the error is returned in **index_returned** and a non−zero value is returned. If an error is generated, a subset of the values may have been changed.

The **UniSetLayoutValues**() function may return one of the following values.

0	All layout values were successfully set.
LAYOUT_EINVAL	The layout value specified by **index_returned** is unknown or its value is invalid or the argument **layout_object** is invalid .
LAYOUT_EMFILE	{OPEN_MAX} files descriptors are currently open in the calling process.
LAYOUT_ENOMEM	Insufficient storage space is available.

8.1.4 UniGetLayoutValues() − Get Layout Values of a LayoutObject

```
#include <UniChar.h>
int UniGetLayoutValues (
        LayoutObject layout_object,
        LayoutValues values,
        int* index_returned
        );
```

The **UniGetLayoutValues**() function is used to query the current setting of layout values within a **LayoutObject**.

The **layout_object** argument specifies a **LayoutObject** returned by the **UniCreateLayoutObject()** function.

The **values** argument specifies the list of layout values (**LayoutValueRec,** see above) that are to be queried but each **value** element of a **LayoutValueRec** must point to a location where the layout value is stored. That is, if the layout value is for type T, the argument must be of type T*. If T itself is a pointer, then **UniGetLayoutValues**() allocates storage to store the actual data, and the caller is responsible for freeing this data by calling **free**(T) with the returned pointer. The values are queried from the **LayoutObject** and represent its current state.

When the **UniGetLayoutValues()** function completes without errors a zero is returned.

If any value can not be queried the index in **values** causing the error is returned in **index_returned** and a non−zero value is returned.

The **UniGetLayoutValues**() function returns one of the following values:

0	All layout values were successfully queried.
LAYOUT_EINVAL	The layout value specified by **index_returned** is unknown or the argument **layout_object** is invalid .
LAYOUT_ENOMEM	Insufficient storage space is available.

8.1.5 UniTransformLayout() − Layout transformation

```
#include <UniChar.h>
int  UniTransformLayout (
        LayoutObject layout_object,     /* LayoutObject Handle */
        const UniChar* InpBuf,          /* Input Text */
        size_t* InpSize,                /* Size of InpBuf − number of UniChar*/
        UniChar* OutBuf,                /* Output Text */
        size_t*  Outsize,               /* Size of OutBuf − number of bytes*/
        size_t *InpToOut,               /* Source To Target Index array */
        size_t *OutToInp,               /* Target To Source Index array */
        unsigned  char *BidiLevel       /* Level array */
        );
```

The **UniTransformLayout()** function transform the **InpBuf** text according to the current layout values in **LayoutObject.** Any layout value whose value type is **LayoutTextDescriptor** describes the attributes of the **InpBuf** and **OutBuf**. If the attributes are the same for both **InpBuf** and **OutBuf** then a null transformation is done with respect to that specific layout value.

The output of this function may be one or more of the following depending on the setting of the respective arguments:

OutBuf/**Outsize**	any transformed data is stored into **OutBuf**.
InpToOut	a cross reference from each **InpBuf** code element to the transformed data.
OutToInp	a cross reference to each **InpBuf** code element from the transformed data.
BidiLevel	a weighted value that represents the directional level of each **InpBuf** code element. The level is dependent on the internal directional algorithm of the **LayoutObject**.

Each of these output arguments may be NULL to specify that no output is desired for the specific argument, but at least one of them should be set to non−NULL to perform any significant work. In addition to the possible output above the **LayoutObject** will maintain a directional state across calls to the transform functions if

1 the *TypeOfText* for **InpBuf** is set to TEXT_EXPLICIT and imbedded directional controls exist, or
2 the code set of the locale is stateful .

The directional state is reset to it's initial state when ever any of the layout values *TypeOfText, Orientation or BidiType* is set via a call to **UniSetLayoutValues**().

The **layout_object** argument specifies a **LayoutObject** returned by the **UniCreateLayoutObject()** function.

The **InpBuf** argument specifies the source text to be processed. The **InpBuf** may not be NULL.

On input, the **InpSize** argument specifies the number of code elements to be processed in units of **UniChar** for **UniTransformLayout**(). A value of −1 indicates that input is delimited by a NULL code element. On return, the value is modified to the actual number of code elements processed in **InpBuf** (unless the value in **OutSize** is zero, in which case, the value of **InpSize** is not changed).

The **OutBuf** argument contains the transformed data. This argument can be specified as a NULL pointer to indicate that no transformed data is required.

The encoding of the **OutBuf** argument depends on the *UniShapeCharset* layout value defined in **layout_object**. If the *ActiveShapeEditing* layout value is not set (False), the encoding of **OutBuf** is guaranteed to be the same as the code set of the locale associated with the **LayoutObject** defined by **layout_object**.

On input, the **OutSize** argument specifies the size of the output buffer in number of **UniChar**. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation will be

performed. If the *ActiveShapeEditing* layout value is set (True) the **OutBuf** is recommended to be allocated to contain at least:

number of code element X *ShapeCharsetSize*.

On return, the **OutSize** argument is modified to the actual number of bytes placed in **OutBuf**.

When the **OutSize** argument is specified as zero, the function will calculate the size of an output buffer large enough to contain the transformed text, and the result will be returned in this field. The content of the buffers specified by **InpBuf** and **OutBuf**, and the value of **InpSize**, will remain unchanged.

If the **InpToOut** argument is not a NULL pointer, it represents an array of values with the same number of code elements as **InpBuf**. On output, the **n**th value in **InpToOut** corresponds to the **n**th code element in **InpBuf**. This value is the index in **OutBuf** which identifies the transformed *UniShapeCharset* element of the **n**th code element in **InpBuf**. **InpToOut** may be specified as NULL if no index array from **InpBuf** to **OutBuf** is desired.

If the **OutToInp** argument is not a NULL pointer, it represents an array of values with the same number of code elements as contained in **OutBuf**. On output, the **n**th value in **OutToInp** corresponds to the **n**th *UniShapeCharset* element in **OutBuf**. This value is the index in **InpBuf** which identifies the original code element of the **n**th *UniShapeCharset* element in **OutBuf**. **OutToInp** may be specified as NULL if no index array from **OutBuf** to **InpBuf** is desired.

If the **BidiLevel** argument is not a NULL pointer, it represents an array of values with the same number of elements as the source text. The **n**th value in **BidiLevel** corresponds to the **n**th code element in **InpBuf**. This value is the level of this code element as determined by the bidirectional algorithm. **BidiLevel** may be specified as NULL if a levels array is not desired.

To perform shaping of a text string without reordering of code elements, the **LayoutObject** should be set with input layout value *TypeOfText* set to TEXT_VISUAL and both **in** and **out** of *Orientation* set to the same value.

If successful, the **UniTransformLayout**() function return a zero.

The **UniTransformLayout()** function may returns one of the following values:

0	The function is completed without errors.
LAYOUT_E2BIG	The output buffer is full and the source text is not entirely processed.
LAYOUT_EINVAL	Transformation stopped due to an incomplete code element or shift sequence at the end of the input buffer. The **InpSize** argument is set to indicate the number of the code element successfully transformed; note, this can be used to determine the code element causing the error.
LAYOUT_ERANGE	More than 15 embedding levels are in source text or **InpBuf** contains unbalanced Directional Format (Push/Pop).

When the size of **OutBuf** is not large enough to contain the entire transformed text, the input text state at the end of the uncompleted transformation is saved internally and the error condition LAYOUT_E2BIG is returned. To resume the transformation on the remaining text, the application should call the **UniTransformLayout()** function with the same **LayoutObject,** the same **InpBuf,** and **InpSize** is set to zero (0).

Example:

The following is an example of transformation of both directional re−ordering and shaping
Note 1 − uppercase represent LTR characters while lowercase represent RTL characters
Note 2 − ”xyz” represent the shapes of ”cde”

```
Position:          0123456789
```

```
InpBuf:          AB cde 12Z

Position:        0123456789
OutBuf:          AB 12 zyxZ

Position:        0123456789
ToTarget:        0128765349

Position:        0123456789
ToSource:        0127865439

Position:        0123456789
BidiLevel:       0001111220
```

Implementation Notes

The values returned in the **BidiLevel** define the directionality of each code element in the source text as defined by the type of algorithm used within the **LayoutObject**. While the algorithm may be implementation dependent the resulting values/levels are commonly defined that allow a single method to be used in determining the directionality of the source text. The base rules are

odd levels are always RTL,
even levels are always LTR, and
Orientation layout value setting determines the initial level (0 or 1) that is used.

Within each a **BidiLevel** array each increment in the level indicates the corresponding code elements should be presented in the opposite direction. Callers of this function should realize that the **BidiLevel** values for certain code elements is dependent on the context of the given character and the layout values: *Orientation &BidiType*. Callers should not assume that a given code element always have the same **BidiLevel** value in all cases.

The following is an example of a standard presentation algorithm that handles nesting correctly. The goal of the algorithm is to ultimately return to a zero state. (NOTE: more efficient algorithms do exist, the following is provided for clarity rather than for efficiency.)

1. Search for the highest level in the string.
2. Reverse all surrounding code elements of the same level. Reduce the level of these code elements by 1.
3. Repeat 1 and 2 until all code elements are of level 0.

The following shows the progression on the example from above:
Note – uppercase represent LTR characters while lowercase represent RTL characters

```
Position:        0123456789        0123456789        0123456789
InpBuf:          AB cde 12Z        AB cde 21Z        AB 12 edcZ
BidiLevel:       0001111220        0001111110        0000000000
```

The values returned from the **BidiLevel** may be used to generate the **InpToOut** and **OutToInp** arrays.

8.1.6 UniEditShapeLayout() – Shape Editing of context dependent text

```
#include <UniChar.h>
int  UniEditShapeLayout (
        LayoutObject layout_object,     /* LayoutObject Handle */
        unsigned int EditType,          /* Type of editing */
        size_t* index,                  /* Cursor Position – in UniChar units */
        const UniChar* InpBuf,          /* Input Text */
        size_t* InpSize,                /* Size of InpBuf  – number of UniChar */
        UniChar* OutBuf,                /* Output Text  */
        size_t* OutSize                 /* Size of OutBuf – number of UniChar*/
        );
```

The **UniEditShapeLayout()** function provides the shapes of the code element pointed to by **index** along with any surrounding code elements as defined by the *ShapeContextSize* layout value of the **LayoutObject** defined by **layout_object**. The **UniEditShapeLayout**() function is intended to be used when performing editing operations on a single code element, e.g. insertion or deletion of a single code element. The **UniTransformLayout**() function should be used when doing string shaping. The **UniEditShapeLayout**() function does not affect any state maintained by the **UniTransformLayout**() function.

This function need not be called if the **front** and **back** elements of the *ShapeContextSize* layout values are both equal to zero (0). If either of these is not zero, then the *ShapeContextSize* layout value specifies the minimum context that needs to be passed into the **UniEditShapeLayout**() function. Yet, these function may use the entire **InpBuf** to determine a wider context.

The **layout_object** argument specifies a **LayoutObject** handle returned by the **UniCreateLayoutObject()** function.

The **EditType** argument specifies the type of edit shaping (*EditInput* or *EditReplace*) that is desired. When **EditType** specifies *EditInput*, it will read the current code element as defined by **index** and any preceding (**front**) code elements as defined by the *ShapeContextSize* layout value. When **EditType** specifies *EditReplace*, it will read the current code element as defined by **index** and any surrounding (**front** and/or **back**) code elements as defined by the *ShapeContextSize* layout value . Note the direction of the editing (i.e. layout values – *Orientation* and *TypeOfText[VISUAL]*) is what determines which are the preceding and which are the succeeding code elements.

When *ActiveShapeEditing* is set (True) the **LayoutObject** maintains an *"EditInput* state" that may affect subsequent calls to the **UniEditShapeLayout**() function with *EditInput* **EditType**. The *EditInput* state is not affected when **EditType** is set to *EditReplace.* To reset the *EditInput* state to its initial state these functions should be called with **InpBuf** set to NULL. The *EditInput* state is not affected if any error occurs within the **UniEditShapeLayout**() functions when **EditType** is *EditInput*.

On input the **index** argument specifies an offset to the start of a code element in **InpBuf** that will be the base for the editing. In addition, the context of surrounding code elements may be considered where the minimum set of code elements needed for the specific context dependent script(s) is identified by the *ShapeContextSize* layout value. If the set of surrounding code elements as defined by **index, InpBuf** and **InpSize** is less then the size of **front**/**back** of the *ShapeContextSize*, the **UniEditShapeLayout**() function will assume there is no additional context available. It is the caller's responsibility to provide the minimum context if available. The **index** argument is in units associated with the type of **InpBuf**, i.e. **UniChar**,

On return, the **index** argument is modified to indicate the offset to the first code element in **InpBuf** that required shaping. The number of code elements that required shaping is indicated on return by the **InpSize** argument.

The **InpBuf** argument specifies the source to be processed. A NULL value with *EditInput* **EditType** indicates a request to reset the *EditInput* state to its initial state.

On input, the **InpSize** argument specifies the number of code elements to be processed. A value of −1 indicates that input is delimited by a NULL code element. On return, the value is modified to the actual number of code elements that needed shaping in **InpBuf**. A value of zero (0) when **EditType** is *EditInput* indicates that the *EditInput* state should be reset to its initial state.

The **OutBuf** argument contains the shaped output text. This argument can be specified as a NULL pointer to indicate that no transformed text is required. If NULL, the function will still return the **index** and **InpSize** that specify the amount of text required to be redrawn.

The encoding of the **OutBuf** argument depends on the *UniShapeCharset* layout value defined in **layout_object**. If the *ActiveShapeEditing* layout value is not set (False), the encoding of **OutBuf** is guaranteed to be the same as the code set of the locale associated with the **layout_object**.

On input, the **OutSize** argument specifies the size of the output buffer in number of **UniChar**. Only the code elements required to be shaped are written into **OutBuf**. The output buffer should be large enough to contain the shaped result; otherwise, only partial shaping will be performed. If the *ActiveShapeEditing* layout value is set (True) the **OutBuf** should be allocated to contain at least:

number of **InpBuf** code element X *ShapeCharsetSize*.

On return, the **OutSize** argument is modified to the actual number of bytes placed in **OutBuf**.

When the **OutSize** argument is specified as zero, the function will calculate the size of an output buffer large enough to contain the transformed text from the input buffer, and the result will be returned in this field. The content of the buffers specified by **InpBuf** and **OutBuf**, and the value of **InpSize**, will remain unchanged.

When the **UniEditShapeLayout**() function completes without errors a zero is returned, **index** and **InpSize** return the minimum set of code elements required to be redrawn, and if **OutBuf** is not NULL the respective shaped code elements are written into **OutBuf**. Any portion of the the **InpBuf** may be indicated as needing to be redrawn/shaped. Otherwise, if an error occurs a non−zero value is returned and any output values depend on the error code.

The **LayoutEditShape**() function may return one of the following values:

0	The function is completed without errors.
LAYOUT_E2BIG	The output buffer is too small and the source text was not processed. **Index** and **InpSize** are not guaranteed on return.
LAYOUT_EINVAL	Shaping stopped due to an incomplete code element or shift sequence at the end of the input buffer. The **InpSize** argument is set to indicate the number of the code element successfully transformed; note, this can be used to determine the code element causing the error.
LAYOUT_ERANGE	More than 15 embedding levels are in source text or **index** is outside the range as defined by **InpSize** or **InpBuf** contains unbalanced Directional Format (Push/Pop).

8.1.7 Layout Values

The following table describes the layout values that shall be common to every locale. These values may be interpreted by the functions handling a **LayoutObject**.

Each value is described in terms of its name, data type and whether the value may be set(**S**) or get (**G**) using **UniSetLayoutValues**() or **UniGetLayoutValues**() respectively.

Value Name	Value Type	SG
ActiveShapeEditing	**BooleanValue**	**G**
ActiveBidirection	**BooleanValue**	**G**
UniShapeCharset	**UniChar***	**SG**
ShapeCharsetSize	**int**	**G**
ShapeContextSize	**LayoutEditSize**	**G**
Orientation	**LayoutTextDescriptor**	**SG**
TypeOfText	**LayoutTextDescriptor**	**SG**
BidiType	**LayoutTextDescriptor**	**SG**
Swapping	**LayoutTextDescriptor**	**SG**
Numerals	**LayoutTextDescriptor**	**SG**
TextShaping	**LayoutTextDescriptor**	**SG**
WordBreak	**LayoutTextDescriptor**	**SG**

The following table describes the layout values that may be supported by some locale. Other locales may ignore these values.

Value Name	Value Type	SG
ArabicSpecialShaping	**LayoutTextDescriptor**	**SG**
ArabicOneCellShaping	**LayoutTextDescriptor**	**SG**

8.1.8 Layout Value Types

The following describe the data types used for some of the layout values. These will be described in

#include <UniChar.h>.

All layout values are defined in this header file. In addition the following layout values may be combined (or'd) into a single **type** "TextDescriptor":

Orientation
TypeOfText
BidiType
Swapping
Numerals
TextShaping
WordBreak.

The **value** of these layout values may also be combined (or'd) into a single attribute. The layout value AllTextDescriptor can be used to indicate that all **LayoutTextDescriptors** will be set (or'd). Refer to example at the beginning of the layout text functions.

The LayoutTextDescriptor is used to identify the attributes of source and target text.

```
typedef struct {
        int in;             /* Input buffer description */
        int out;            /* Output buffer description */
        } LayoutTextDescriptorRec, *LayoutTextDescriptor
```

The **inp** and **out** values are corresponding combinations of the appropriate descriptor items. Each of the descriptors is specified as a combination of one value from each of the following groups −− a value for the input descriptor and a corresponding value for the output descriptor.

The LayoutEditSize structure defines the number of surrounding code elements that need to be considered when performing edit shaping, i.e. calling the **UniEditShapeLayout**() function.

```
typedef struct {
        int front;          /*# of code element in front of the edit position, logical order */
        int back;           /*# of code elements following the edit position, logical order*/
        } LayoutEditSizeRec, *LayoutEditSize;
```

When a substring is inserted into a string, the **front** and **back** elements define the number of code elements in front of the substring and the number of code elements after the substring, respectively, that need to be considered when performing edit shaping. The total number of code elements needed to be viewed is

front + # code elements in substring + **back**.

If both **front** and **back** elements are set to zero then no additional context needs to be considered for edit shaping. When *ActiveShapeEditing* is not set (False) , the **front** and **back** are guaranteed to be zero (0).

8.1.9 ActiveBidirection

If the descriptor **ActiveBidrection** is set (True) then the **LayoutObject** includes knowledge of directional code elements and proper rendering of text will require re−ordering of directional code elements. Otherwise the **LayoutObject** does not require any re−ordering of directional code elements and all code elements will be classified as left−to−right.

The **ActiveBidirection** value is guaranteed to not change for the life of the **LayoutObject**.

8.1.10 ActiveShapeEditing

If the descriptor **ActiveShapeEditing** is set to True then the **LayoutObject** includes knowledge of context dependent code elements (i.e. an Automatic Shape Determination algorithm) that require shaping for presentation to the **UniShapeCharset**. If True the user of a **LayoutObject** is required to perform some transformation and/or edit shaping prior to rendering the text. (NOTE: for tty based applications the shape editing may be assumed to be done at the terminal.)

Otherwise, the **LayoutObject** does no shaping and all code elements may be presented independent of the surrounding characters.

The **ActiveShapeEditing** value is guaranteed to not change for the life of the **LayoutObject**.

8.1.11 UniShapeCharset

For the **ULS** the **UniShapeCharset** will always be **"UCS−2**".

The descriptor **UniShapeCharset** specifies the encoding that will be used when performing **ActiveShapeEditing**. If **ActiveShapeEditing** is not set (False) the **UniShapeCharset** is guaranteed to match the code set associated with the locale of the **LayoutObject**, i.e. shape editing is a null operation.

A charset is defined (by Xlib specification) as "an encoding with a uniform, state−independent mapping from character to code points, or a coded character set" and is usually used to identify the encoding of the glyphs within a font. A **UniShapeCharset** is a well known name associated with some type of presentation encodings and it is expected that the **UniShapeCharset** names will be registered with the X Consortium's registry for CHARSET's. Yet, a **UniShapeCharset** need not be a font encoding but may be some intermediate encoding that can then be rendered to a specific font. (**Note:** some **LayoutObject** may be extended to provide a layout value by which the per glyph metrics/dimensions may be passed into it. Such extensions are outside the scope of this specification.)

Since the **UniShapeCharset** is associated with a font/glyph encoding, it may not be the same as the code set of the locale associated with the **LayoutObject**. The **UniShapeCharset** is guaranteed to be of a uniform size and state−independent but the size of each **UniShapeCharset** may vary (e.g. 8/16/other bits) so applications should use the **ShapeCharsetSize** value when doing storage management.

If the **UniShapeCharset** is set to the **LayoutCharsetIsWchar** reserved string then shaping will be done into the **wchar_t** encoding.

8.1.12 ShapeCharsetSize

The descriptor **ShapeCharsetSize** specifies the encoding size of the current **UniShapeCharset**. This value may change when the **UniShapeCharset** is changed. If **ActiveShapeEditing** is not set (False) the **ShapeCharsetSize** is set to the maximum code element size (in bytes) per the code set of the locale for the **LayoutObject**.

8.1.13 ShapeContextSize

The **ShapeContextSize** specifies the size of the context (surrounding code elements) that needs to be accounted for when performing **ActiveShapeEditing**. The **ShapeContextSize** is defined as structure of type **LayoutEditSize** and is described above.

The **ShapeContextSize** value is guaranteed to not change for the life of the **LayoutObject**.

8.1.14 Orientation

The descriptor **Orientation** specifies the global directional text orientation:

ORIENTATION_CONTEXTUAL − contextual global orientation

ORIENTATION_LTR − left−to−right global orientation

ORIENTATION_RTL − right−to−left global orientation

When ORIENTATION_CONTEXTUAL is set, the **Orientation** setting is determined according to the direction of the first significant code element. The initial state for **Orientation** is locale dependent.

8.1.15 TypeOfText

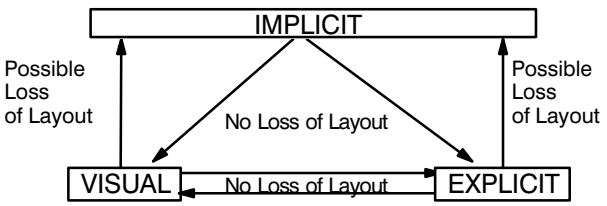
The **TypeOfText** descriptor specifies the directional ordering of the directional text in the buffer:

TEXT_VISUAL − code elements are stored in visual ordered segments which can be rendered as is. Bidirectional code elements will have be previously re−ordered for a specific orientation.

TEXT_IMPLICIT − code elements are stored in logically ordered segments. Any directional controls, if defined by the code set of the locale, are ignored. Re−ordering of directional code elements is defined by the **BidiType** layout value.

TEXT_EXPLICIT − code elements are stored in logically ordered segments with a set of embedded controls as defined by the **BidiType** layout value. For this **TypeOfText** the **LayoutObject** will preserve a *bidirectional state* across calls to the **UniTransformLayout**() function. The directional state is reset to the initial state each time **TypeOfText** is set to any value. Re−ordering of directional code elements is defined by the **BidiType** layout value.

Each **LayoutObject** shall be expected to provide transformation from each of the above type to any of the other types. Yet, some transformations may cause layout (directional) information to be lost such that text is presented differently after a round trip transformation. For this case it does not imply any data loss, just loss in layout information. The following illustrates where layout information may be lost:



8.1.16 BidiType

The **BidiType** descriptor will specify the type of implicit algorithm and the set of embedded controls used in re−ordering and shaping of directional/context dependent text:

BIDI_DEFAULT − Bidirectional code elements should be re−ordered using an implementation defined implicit bidirectional algorithm when converting to/from an implicit form.

The set of embedded controls for TEXT_EXPLICIT is implementation defined. The following is an example of the ISO 6429 controls:

- Begin left−to−right
- Begin right−to−left
- End of String

The encoding of the above controls depends on the code set associated with the **LayoutObject**. Any ASCII based encoding will use the ISO 6429 escape sequence definitions.

BIDI_UCS −Bidirectional code elements should be re−ordered using the Unicode implicit bidirectional algorithm when converting to/from an implicit form. The set of embedded controls are defined by Unicode.

The following controls will be recognized when re−ordering with TEXT_IMPLICIT:

LEFT−TO−RIGHT MARK (LRM)
RIGHT−TO−LEFT MARK (RLM)

The following controls will be recognized when re−ordering with TEXT_EXPLICIT:

LEFT−TO−RIGHT EMBEDDING (LRE)
RIGHT−TO−LEFT EMBEDDING (RLE)
RIGHT−TO−LEFT OVERRIDE (LRO)
LEFT−TO−RIGHT OVERRIDE (RLO)
POP DIRECTIONAL FORMAT (PDF)

In addition, the following controls will affect the **LayoutObject** values associated with the following code elements of UCS whenTEXT_EXPLICIT is set with BIDI_UCS:

INHIBIT SYMMETRIC SWAPPING
ACTIVATE SYMMETRIC SWAPPING
INHIBIT ARABIC FORM SHAPING
ACTIVATE ARABIC FORM SHAPING
NATIONAL DIGIT SHAPES
NOMINAL DIGIT SHAPES

For **ULS**, the BIDI_DEFAULT will be the same as BIDI_UCS.

8.1.17 Swapping

The **Swapping** descriptor specifies whether symmetric swapping is applied to the text:

SWAPPING − The text conforms to symmetric swapping

NO_SWAPPING − The text does not conform to symmetric swapping

8.1.18 Numerals

The **Numerals** descriptor specifies the shaping of numerals recognized by the **LayoutObject**:

NUMERALS_NOMINAL – Nominal shaping of numeral using the portable character set (i.e. Arabic numerals).

NUMERALS_NATIONAL – National shaping of numeral based on the script of the locale associated with the **LayoutObject** (e.g. the script used may be Thai, Farsi, Hindi, or Bengali numerals).

NUMERALS_CONTEXTUAL – Contextual shaping of numeral, depending on the context (script) of surrounding text (e.g. Hindi numbers in Arabic text and Arabic numbers otherwise). At a minimum this option will be assured to support and recognize text whose script is defined by the NUMERALS_NATIONAL. Any other script is implementation dependent.

8.1.19 Text Shaping

TextShaping: specifies the characteristics of the **in**/**out** text.

TEXT_SHAPED – The text contains presentation form shapes as defined by the locale

TEXT_NOMINAL – The text has no presentation form shapes as defined by the locale.

TEXT_INITIAL – The visual text has only initial shapes.

TEXT_MIDDLE – The visual text has only middle shapes.

TEXT_FINAL – The visual text has only final shapes.

TEXT_ISOLATED – The visual text has only isolated shapes.

The **in** set of shaping characters is limited to the code set of the locale associated with the **LayoutObject**.

8.1.20 ArabicSpecialShaping

The **ArabicSpecialShaping** descriptor relates to the number of code elements used in special cases when swapping text:

TEXT_STANDARD – The text contains standard presentation form shapes as defined by the locale

TEXT_SPECIAL – The text has special shapes that are locale defined.

The **UniShapeCharset** descriptor is ignored if ArabicSpecialShaping is set to TEXT_SPECIAL.

For example, the Arabic locale recognizes the following special shapes differences:

TEXT_STANDARD	TEXT_SPECIAL
LAMALEF is represented using 2 code points.	LAMALEF is shaped using one code point
SEEN like code elements may be represented as one or two code points.	SEEN like code element are represented using two code points.
Isolated and final YEH_HAMZA is represented as a single code point.	YEH_HAMZA is represented as 2 code points.

8.1.21 ArabicOneCellShaping

The **ArabicOneCellShaping** descriptor relates to the number of graphic columns used for representation of SEEN−like code elements (SEEN−SHEEN−SAD−DAD) in its final or isolated form. A tail could be added if there is space for it, otherwise a single code point is used.

ONECELL − SEEN−like code elements are encoded to occupy a single display column.

TWOCELL − SEEN−like code elements are encoded to occupy 2 display columns.

This descriptor is ignored if **ArabicSpecialShaping** is not set to TEXT_SPECIAL

8.1.22 WordBreak

The **WordBreak** descriptor specifies whether bidirectional reordering should factor word recognition or not.

BREAK − word break on, word by word reordering (i.e. the text reordering is dependent on the level (per the birectional algorithm) of word separator).

NO_BREAK − word break off, segment reordering.

This descriptor is significant only if **TypeOfText** is TEXT_IMPLICIT.

9 Conversion Functions

9.1 Conversion Overview

It is expected that most of the processing by applications using the ULS will be done using ***UniChar**** strings. Yet, many applications will need to export and/or import data to/from non−UCS encodings (e.g. ASCII or EBCDIC). For this purpose, a set of functions are defined to perform conversions between UCS and non−UCS encodings. The **Uconv** name is used to indicate these functions. The **Uconv** functions are only capable of doing *UCS conversions,* i.e. convert to/from UCS.

In addition, the **iconv** functions defined by X/Open Portability Guide (XPG) Issue 4 are provided for converting between any encodings, e.g. between **PC** code and **ISO** codes. In addition, any of the *UCS conversions* may be invoked using the **iconv** functions by specifying **UCS−2** as the to/from code set.

Several conversions specific to various regions of the world are provided as optional, installable packages. Refer to **List of Conversions** in the **ULS Localization** appendix for a complete list of conversion names available. It should be noted that the set of conversions available at run time depend on those that were installed.

Refer to **Conversion Modifiers** for a list of possible conversion options.

Refer to the **uconvdef** utility for how to customize *UCS conversions*.

9.2 Universal Conversion Functions (Uconv)

9.2.1 UniCreateUconvObject() − Open and initialize a Uconv object

```
#include <uconv.h>
int UniCreateUconvObject(
        UniChar *code_set,              /* Specifies the name of the UCS conversion. */
        UconvObject *uconv_object       /* Specifies the conversion object being returned. */
        );
```

The **UniCreateUconvObject**() function returns a conversion object that describes a *UCS conversion* between the code set specified by **code_set** and **UCS**. For state−dependent encodings, the conversion object will be in a code set dependent initial shift state, ready for immediate use with either the **UniUconvToUcs**() or **UniUconvFromUcs**().

A conversion object remains valid in a process until that process closes it.

Settings of **code_set** are implementation−dependent. However, the **UniCreateUconvObject**() function will support a grammar of **modifiers** used to augment code set names. This grammar permits the passing of arguments to the conversion objects and is described in section 9.4.

The **UniCreateUconvObject*()*** uses the **LOCPATH** environment variable to search for a loadable conversion file which is used to carry out conversions between **code_set** and **UCS−2**. This file will typically be a table generted using the **uconvdef** utility (section 10.3.2).

The **UniCreateUconvObject**() returns a conversion object in **uconv_object** for use in subsequent calls to either the **UniUconvFromUcs**() or **UniUconvToUcs**() functions.

Return Value

Upon completion, the **UniCreateUconvObject*()*** function will return one of the following values:

0	Conversion object successfully initialized.
UCONV_EMFILE	{OPEN_MAX} file descriptors are currently open in the calling process
UCONV_ENFILE	Too many files currently open in the system
UCONV_ENOMEM	Insufficient storage space is available
UCONV_EINVAL	The conversion specified by **code_set** or the modifier supplied is not recognized by the implementation. Note, some conversions need to be installed before they can be used.

9.2.2 UniUconvToUcs() − Convert from Multibyte to UCS

```
#include <uconv.h>
int UniUconvToUcs(
        UconvObject uconv_object,       /* Uconversion object handle */
        void** inbuf,                   /* Input buffer. */
        size_t* inbytesleft,            /* Size of inbuf, in units of bytes */
        UniChar** ucsbuf                /* Output buffer */
        size_t** UniCharsleft,          /* Size of ucsbuf */
        size_t* nonidentical            /* number of non identical conversions */
        );
```

The **UniUconvToUcs***(*) function converts a sequence of characters encoded in one code set, in the array specified by **inbuf**, into a sequence of corresponding **UCS** code elements, in the array specified by **ucsbuf**. The code set of the **inbuf** is the string specified in the **UniCreateUconvObject**() call that returned the conversion object, **uconv_object**. The **inbuf** argument points to a variable that points to the first byte in the input buffer and the **inbytesleft** indicates the number of bytes to the end of the buffer to be converted. The **ucsbuf** argument points to a variable that points to the first available ***UniChar*** in ucsbuf and **UniCharsleft** indicates the number of ***UniChar*** elements available to the end of the buffer.

If the **UniCharsleft** argument specifies a zero value, **UniUconvToUCS**() will calculate the number of bytes required to contain the converted string and write it into **UniCharsleft**. The content of **ucsbuf** and **inbuf** are not not modified if **UniCharsleft** is zero.

For state−dependent encodings, the conversion object **uconv_object** is placed into its initial shift state by a call for which **inbuf** is a null pointer, or for which **inbuf** points to a null pointer. When **UniUconvToUCS**() is called in this way, and if **ucsbuf** is not a null pointer or a pointer to a null pointer, and **UniCharsleft** points to a positive value, **UniUconvToUCS**() will place, into **ucsbuf** the ***UniChar*** sequence to change the **ucsbuf** to its initial shift state. If the **ucsbuf** is not large enough to hold the entire reset sequence, **UniUconvToUCS**() will fail [UCONV_E2BIG]. Subsequent calls with **inbuf** as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion object.

If a sequence of bytes within **inbuf** does not form a valid character in the specified code set, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the **ucsbuf** buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by **inbuf** is updated to point to the byte following the last byte successfully used in the conversion. The value pointed to by **UniCharsleft** is decrement to reflect the number of ***UniChar*** elements still available in **ucsbuf**. For state−dependent encodings, the conversion object is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

If **UniUconvToUCS**() encounters a character in the **inbuf** that is legal, but for which an identical character does not exist in UCS, **UniUconvToUCS**() performs an implementation−defined conversion on this character.

Return Value

The **UniUconvToUCS**() function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns in **nonidentical** the number of non−identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by **inbytesleft** will be zero. If the input conversion is stopped due to any condition mentioned above, the value pointed to by **inbytesleft** will be non−zero and a non−zero value is returned to indicate the condition. If an error occurs, **UniUconvToUCS**() returns a non−zero value.

The **UniUconvToUCS**() function will return one of the following values:

0	Conversion successful.

UCONV_EILSEQ	Input conversion stopped due to an input byte that does not belong to the input code set.
UCONV_E2BIG	Input conversion stopped due to lack of space in the output buffer.
UCONV_EINVAL	Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer.
UCONV_EBADF	The **uconv_object** argument is not a valid open conversion object.

9.2.3 UniUconvFromUcs() − Convert UCS to Multibyte

```
#include <uconv.h>
int UniUconvFromUcs(
        UconvObject uconv_object,       /* Specifies the Uconv object. */
        UniChar** ucsbuf,               /* Input buffer */
        size_t* UniCharsleft,           /* Size of ucsbuf. */
        void** outbuf,                  /* Output buffer. */
        size_t* outbytesleft,           /* Size of outbuf in units of bytes */
        size_t* nonidentical            /* number of non identical conversions */
        ):
```

The **UniUconvFromUCS**() function converts a sequence of code elements, in the array specified by **ucsbuf**, into a sequence of corresponding characters in another code set, in the array specified by **outbuf**. The code set of the **outbuf** is the string specified in the **UniCreateUconvObject**() call that returned the conversion object, **uconv_object**. The **ucsbuf** argument points to a variable that points to the first ***UniChar*** in the input buffer and the **UniCharsleft** indicates the number of ***UniChar*** elements to the end of the buffer to be converted. The **outbuf** argument points to a variable that points to the first available character in **outbuf** and **outbytesleft** indicates the number of bytes available to the end of the buffer.

If the **outbytesleft** argument specifies a zero value, **UniUconvFromUCS**() will calculate the number of bytes required to contain the converted string and write it into **outbytesleft**. The content of **ucsbuf** and **outbuf** are not modified if **outbytesleft** is zero.

For state−dependent encodings, the conversion object **uconv_object** is placed into its initial shift state by a call for which **ucsbuf** is a null pointer, or for which **ucsbuf** points to a null pointer. When **UniUconvFromUCS**() is called in this way, and if **outbuf** is not a null pointer or a pointer to a null pointer, and **outbytesleft** points to a positive value, **UniUconvFromUCS**() will place, into **outbuf** the byte sequence to change the **outbuf** to its initial shift state. If the **outbuf** is not large enough to hold the entire reset sequence, **UniUconvFromUCS**() will fail [UCONV_E2BIG]. Subsequent calls with **ucsbuf** as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion object.

If a sequence of ***UniChar*** elements within **ucsbuf** does not form a valid code element, conversion stops after the previous successfully converted ***UniChar***. If the **ucsbuf** buffer ends with an incomplete code element or shift sequence, conversion stops after the previous successfully converted ***UniChar*** elements. The latter will occur when the **ucsbuf** buffer ends with a combining character. If the end of a stream ends with a combining character, the caller should call **UniUconvFromUCS**() with **ucsbuf** set to a null pointer and **outbuf** pointing to a pointer of a non−null buffer. This will cause the *UCS conversion* to write the last code element.

If the **outbuf** buffer is not large enough to hold the entire converted input, conversion stops just prior to the input ***UniChar*** that would cause the output buffer to overflow. The variable pointed to by **ucsbuf** is updated to point to the ***UniChar*** following the last ***UniChar*** successfully used in the conversion. The value pointed to by **outbytesleft** is decrement to reflect the number of bytes still available in **outbuf**. For state−dependent encodings, the conversion object is updated to reflect the shift state in effect at the end of the last successfully converted code element sequence.

If **UniUconvFromUCS**() encounters a code element in the **ucsbuf** that is legal, but for which an identical character does not exist in the target code set, **UniUconvFromUCS**() performs an implementation−defined conversion on the code element.

Return Value

The **UniUconvFromUCS**() function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns in **nonidentical** the number of non−identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by **UniCharsleft** will be zero. If the input conversion is stopped due to any condition mentioned above, the value pointed to by **UniCharsleft** will be non−zero. If any error occurs **UniUconvToUcs**() returns a non−zero value.

The **UniUconvFromUCS**() function will return one of the following values

0	Conversion successful
UCONV_EILSEQ	Input conversion stopped due to an input byte that does not belong to the input code set.
UCONV_E2BIG	Input conversion stopped due to lack of space in the output buffer.
UCONV_EINVAL	Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer. This condition will be given when the last ***UniChar*** element is recognized as a combining character.
UCONV_EBADF	The **uconv_object** argument is not a valid open conversion object.

9.2.4 UniFreeUconvObject() – Close and free memory of a UCS conversions object

```
#include <uconv.h>
int UniFreeUconvObject(
                UconvObject uconv_object                /* Specifies the conversion object */
                );
```

The **UniFreeUconvObject**() function closes the conversion object co and frees any associated structures.

Return Value

Upon completion, the **UniFreeUconvObject*()*** function will return one of the following values:

0	All resources associated with **uconv**
UCONV_EBADF	The **uconv_object** argument is not a valid open conversion object.

9.3 International Conversion Functions (ICONV)

The following specification is based on the **XPG4** specification. It describes a set of functions for performing conversions between any pair of encodings. Since **UCS−2** represents a virtually complete and consistent representation of all characters represented by all encodings, it is provides the perfect interchange format for performing any conversion between these encodings. A high degree of interoperability can be obtained quite economically by implementing **UCS−2** converters (section 9.2) for all encodings of interest, and then implementing the following functions to use **UCS−2** as an interchange medium for performing *any−to−any* conversions.

The following functions can also support conversion to/from **UCS−2** as a code set.

9.3.1 iconv_open() − code conversion allocation function

```
#include <iconv.h>
iconv_t iconv_open(
        const char* tocode,          /* Specifies the encoding of the  input string. */
        const char* fromcode         /*Specifies the encoding of the output string */
        );
```

The **iconv_open***()* function returns a conversion descriptor that describes a conversion from the codeset specified by the string pointed to by the **fromcode** argument to the codeset specified by the string pointed to by the **tocode** argument. For state−dependent encodings, the conversion descriptor will be in a codeset−dependent initial shift state, ready for immediate use with the **iconv**() functions.

Settings of **fromcode** and **tocode** and their permitted combinations are implementation−dependent. The **fromcode** or **tocode** may specifies **UCS−2**.

A conversion descriptor remains valid in a process until that process closes it.

Return Value

Upon successful completion **iconv_open***()* returns a conversion descriptor for use on subsequent calls to **iconv**(). Otherwise **iconv_open**() returns (*iconv_t*)−1 and sets errno to indicate the error.

EMFILE	{OPEN_MAX} file descriptors are currently open in the calling process
ENFILE	Too many files currently open in the system
ENOMEM	Insufficient storage space is available
EINVAL	The conversion specified by **tocode** and **fromcode** is not recognized by the implementation. Note, some conversions need to be installed before they can be used.

9.3.2 iconv() − Perform code set conversion

```
#include <iconv.h>
size_t iconv(
        iconv_t cd,                 /* Specifies the conversion descriptor. */
        char **inbuf,               /* Specifies the input buffer. */
        size_t *inbytesleft,        /* Specifies the bytes left in the input buffer. */
        char **outbuf,              /* Specifies the output buffer. */
        size_t *outbytesleft        /* Specifies the bytes left in the output buffer. */
        );
```

The **iconv***()* function converts the sequence of characters from one codeset, in the array specified by **inbuf**, into a sequence of corresponding characters in another codeset, in the array specified by outbuf. The codesets are those specified in the **iconv_open***()* call that returned the conversion descriptor, ***cd***. The **inbuf** argument points to a variable that points to the first character in the input buffer and **inbytesleft** indicates the number of bytes to the end of the buffer to be converted. The **outbuf** argument points to a variable that points to the first available byte in the output buffer and **outbytesleft** indicates the number of the available bytes to the end of the buffer.

For state−dependent encodings, the conversion descriptor **cd** is placed into its initial shift state by a call for which **inbuf** is a null pointer, or for which **inbuf** points to a null pointer. When **iconv**() is called in this way, and if **outbuf** is not a null pointer or a pointer to a null pointer, and **outbytesleft** points to a positive value, **iconv**() will place, into the output buffer, the byte sequence to change the output buffer to its initial shift state. If the output buffer is not large enough to hold the entire reset sequence, **iconv***()* will fail and set *errno* to [E2BIG]. Subsequent calls with **inbuf** as other than a null pointer or a pointer to a null pointer cause the conversion to take place from the current state of the conversion descriptor.

If a sequence of input bytes does not form a valid character in the specified codeset, conversion stops after the previous successfully converted character. If the input buffer ends with an incomplete character or shift sequence, conversion stops after the previous successfully converted bytes. If the output buffer is not large enough to hold the entire converted input, conversion stops just prior to the input bytes that would cause the output buffer to overflow. The variable pointed to by **inbuf** is updated to point to the byte following the last byte successfully used in the conversion. The value pointed by **inbytesleft** is decrement to reflect the number of bytes still not converted in the input buffer. The variable pointed to by **outbuf** is updated to point to the byte following the last byte of converted output data. The value pointed to by **outbytesleft** is decrement to reflect the number of bytes still available in the output buffer. For state−dependent encodings, the conversion descriptor is updated to reflect the shift state in effect at the end of the last successfully converted byte sequence.

Return Value

If **iconv***()* encounters a character in the input buffer that is legal, but for which an identical character does not exist in the target codeset, **iconv***()* performs an implementation−defined conversion on this character.

The **iconv**() function updates the variables pointed to by the arguments to reflect the extent of the conversion and returns the number of non−identical conversions performed. If the entire string in the input buffer is converted, the value pointed to by **inbytesleft** will be zero. If the input conversion is stopped due to any conditions mentioned above, the value pointed to by **inbytesleft** will be non−zero and *errno* is set to indicate the condition. If an error occurs **iconv**() returns *(size_t)−1* and sets *errno* to indicate the error.

The **iconv**() function will fail if:

EILSEQ	Input conversion stopped due to an input byte that does not belong to the input code set.
E2BIG	Input conversion stopped due to lack of space in the output buffer.

EINVAL Input conversion stopped due to an incomplete character or shift sequence at the end of the input buffer. When converting from **UCS**, this condition will be given when the last ***UniChar*** element is recognized as a combining character.

The **iconv**() function may fail if:

EBADF The **cd** argument is not a valid open conversion descriptor.

9.3.3 iconv_close() − Close and free memory of a conversion descriptor

```
#include <iconv.h>
int iconv_close(
        iconv_t cd                      /* Conversion Descriptor */
        );
```

The **icon_close**() function closes the conversion descriptor **cd** and frees any associated structures.

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.

The **iconv_close**() function may fail if:

EBADF The **cd** argument is not a valid open conversion descriptor.

9.3.4 iconv_query() − Query for possible conversions

```
#include <iconv.h>
int iconv_query(
        const char *code_set,           /* Code set to query */
        char ***to_list,                /* list of valid conversions to code_set */
        char ***from_list               /* list of valid conversions from code_set */
        );
```

The **iconv_query()** function queries for possible conversions to and from the code set identified by the **code_set** argument. The lists of possible code sets are returned in the **from_list** and **to_list** arguments. Either of these may be **NULL** to indicate that the respective **_list** is not desired. If they are not set to **NULL**, the function will return a **NULL**−terminated array of code set names (strings) in each of them.

For example, to find all conversions for ISO8859−1, the **code_set** is set to **"ISO8859−1"** and the **to_list** and **from_list** arguments are set to addresses of valid **char **** variables. In this case, the **to_list** will contain the list of code sets that can be converted **to** ISO8859−1, while the **form_list** will contain the list of code sets that can be converted **from** ISO8859−1.

The search for possible conversions is implementation dependent and any pattern matching is also system dependent. If a code set name is returned in either **from_list** and/or **to_list**, then the **iconv_open**() function will be successfull if given that name and **code_set** as arguments.

Return Value

Upon successful completion, a **0** is returned, otherwise **−1** is returned. If successful, **from_list** and **to_list** are set to point to the lists of possible conversions.

9.4 Conversion Modifiers

The **UniCreateUconvObject**() function will support a grammar for ”augmented code set names” that allow arguments to be passed to conversions objects. The arguments are used to modify the default conversion. A list of common modifiers along with their values and definition are provided below, yet, it is left up to each conversion to decide which modifiers it will recognize.

The grammar for specifying augmented code set names is:

codeset_name : codeset
| codeset’@’argument_list
;

codeset : registry’_’encoding
;

argument_list : argument
| argument’,’argument_list
;

argument : arg_name’=’arg_value
;

registry : PCSstring
;

encoding : PCSstring
;

arg_name : PCSstring
;

arg_value : string
;

string : PCSstring
| ’\’digits
| ’\x’hexdigits
;

hexdigits : digits
| ’[A−F]’
| ’[a−f]’
;

digits : ’[0−9]’
;

multibytestring : any coded character encoded in a multibyte code
;

PCSstring : any character defined in the Portable Character Set
;

The set of *arg_name*s is dependent on the implementation of the conversion and the locale object being used. If no *argument_list* is provided a default conversion is done.

The table in Figure 7 describes the conversion modifiers that may be used to tailor conversion depending on locale being used.

Argument	Locale Req.	Name	Value
Locale	All	locale=	PCSstring
Substitution	N/A	sub=	"**no**" "yes" "to−ucs" "from−ucs"
Substitute Char	N/A	subchar=	*arg_value*
Orientation	ar_AA, iw_IL	orient=	"**context**" "ltr" "rtl"
Type of Text	ar_AA, iw_IL	type=	"**logical**" "implicit" "visual" "control"
Swapping	ar_AA, iw_IL	swap=	"**no**" "yes"
Numerals	ar_AA, iw_IL	num=	"**nominal**" "national" "context"
Text Shaping	ar_AA, iw_IL	shape=	"**nominal**" "shaped" "initial" "middle" "final" "Isolated"
Endian	N/A	endian=	"**system**" "big" "little"

Figure 7. Conversion modifiers, argument names and values

9.4.1 Locale Argument

The *locale*= argument specifies the locale that the conversion object will use to acquire locale dependent information. If this argument is not specified, it is implementation dependent whether any and which locale object will be used.

Currently, the only set of locale dependent information is the set of Bi−directional transforms needed. See **UniCreateLayoutObject**() for more information.

9.4.2 Substitute Argument

The *sub=* argument specifies to the conversion object how to perform in cases that there is no identical character for a given code element. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and ***char**** for **iconv_open**(). The values may be any of the following strings:

"to−ucs"	Specifies that substitution takes place when a sequence of bytes within multibyte buffer does not form a valid character in the specified code set. The initial byte of the unrecognized byte sequence is converted into the *UCS substitution character* (U+001A) and conversion continues starting at the second byte.
"from−ucs"	Specifies that substitution takes place when a code element can not be converted into an identical character in the specified code set. The code element is converted into a multibyte character using a locale defined substitution character or the "*sub−char*" argument if one is specified.
"yes"	Substitution is allowed for either conversion direction as specified in **to−ucs** and **from−ucs**. All conversion are expected to recognize this value.
"no"	Substitution of invalid characters is not allowed in either conversion direction. The conversion process terminates at the first character that requires substitution. All conversions are expected to recognize this value.

9.4.3 Substitute Character Argument

The *subchar= argument* specifies to a conversion object the multibyte character to be used as the substitute character when converting from UCS to a multibyte string. This argument is not expected to be recognized by any locale object. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and ***char**** for **iconv_open**(). Any multibyte string will need to be specified using hex ('\xhexdigits') or digit ('\digits') notation.

If this value is not specified, an implementation defined substitute character will be used.

9.4.4 Orientation Argument

The *orient= s*pecifies the orientation of the input and output strings.

This argument is recognized only by the converter. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and ***char**** for **iconv_open**().

The value string will be of the syntax "%in:%out" where %in and %out specify the ordering for the input and output strings respectively. The %in and %out may specify one of the following:

"**context**"	Contextual global orientation
"**ltr**"	LeftToRight global orientation
"**rtl**"	RightToLeft global orientation

Refer to "**Orientation**" in the function **UniCreateLayoutObject**().

9.4.5 Type of Text Argument

The *type=* argument specifies the type of text expected to be converted. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and char* for **iconv_open**().

The value string will be of the syntax "%in:%out" where %in and %out specify the bidirectional type of text for the input and output strings respectively. The %in and %out may specify one of the following:

"**visual**"	Visual ordering

"**implicit**"	Implicit ordering (this value is applicable only to multibyte codeset text descriptors)
"**logica**l"	Logical ordering (this value is applicable only to Unicode text descriptors; the Bidi Unicode text has no explicit Bidi directional controls)
"**control**"	Controlled ordering (this value is applicable only to Unicode text descriptors; the Bidi Unicode text can have any Bidi directional controls)

Refer to the **TypeOfText** layout value in the function **UniCreateLayoutObject**().

9.4.6 Swapping Argument

The *swap*= argument specifies whether symmetric swapping is applied to the text:

The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and char* for **iconv_open**().

The value string will be of the syntax "%in:%out" where %in and %out specify the type of swapping for the input and output strings respectively. The %in and %out may specify one of the following:

"**yes**"	The text conforms to symmetric swapping
"**no**"	The text does not conform to symmetric swapping

Refer to the **Swapping** layout value in the function **UniCreateLayoutObject**().

9.4.7 Numerals Argument

The *num= argument s*pecifies the shaping of the European numerals:

This argument is not expected to be recognized by any locale objects. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and char* for **iconv_open**().

The value string will be of the syntax "%in:%out" where %in and %out specify the numeral respresentation for the input and output strings respectively. The %in and %out may specify one of the following:

"**nominal**"	Nominal shaping of European numerals and using European numbers
"**national**"	National shaping of European numerals and using Hindi numbers
"**context**"	Contextual shaping of European numerals, depending on the context of the surrounding text (Hindi numbers in Arabic text and European numbers otherwise).

Refer to the **Numerals** layout value in the function **UniCreateLayoutObject**().

9.4.8 Text Shaping Argument

The *shape*= specifies whether Arabic text is shaped:

The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and char* for **iconv_open**().

The value string will be of the syntax "%in:%out" where %in and %out specify the text shaping for the input and output strings respectively. The %in and %out may specify one of the following:

"**nominal**"	The text has no Arabic presentation form shapes
"**shaped**"	The text contains Arabic presentation form shapes

Refer to the **TextShaping** layout value in the function **UniCreateLayoutObject**().

9.4.9 Endian Argument

The *endian*= specifies the endian desired when converting to/from **UCS**.

The value string will be of the syntax ”%in:%out” where %in and %out specify the endian for the input and output strings respectively. The value of this argument is of type ***UniChar**** for **UniCreateUconvObject**() and char* for **iconv_open**(). The %in and %out may specify one of the following:

”**system**”	A octet encoded **UCS** strings will be treated as being in endian of the system, i.e. ***UniChar***’s endian. This is the default.
”**big**”	Octet encoded **UCS** strings will be treated as being in big endian
”**little**”	Octet encoded **UCS** strings will be treated as being in little endian

When used with **UniCreateUconvObject**(), the endian of **UniChar** is always ”**system**” and can not be overridden.

9.5 Managing Coded Character Set Identifiers For Interoperability

Historically, code set and code pages have been identified differently across various systems. This document will focus on usage of **UCS** as the universal code used for both local processing and as the one common network encoding that may be used for interchange. To support applications that desire to import and export from various environments, the **ULS** will support 2 forms of code identification.

typedef unsigned long **ucsid;** /*** *Universal Code Set Id* ***/

A 32 bit integer value that will be used to represent coded character sets. The values will be based on the proposed OSF "*Code Set Registry*". This registry will be maintained by OSF, and is expected to include the IBM **CDRA** registry's **CCSID**s.

UniChar*

Each **ucsid** will have a corresponding name that can be used with the **iconv_open**() function and or **UniCreateUconvObject**() function. For the purposes of this chapter, the term "*code set name*" is used to apply to the names of *code sets, code pages, ccsid's, character sets*.

All code set names in string form will conform to the "**Code Set Naming Conventions**" as defined in the **ULS Localization** appendix. Figure 8 illustrates the mappings that will be provided:

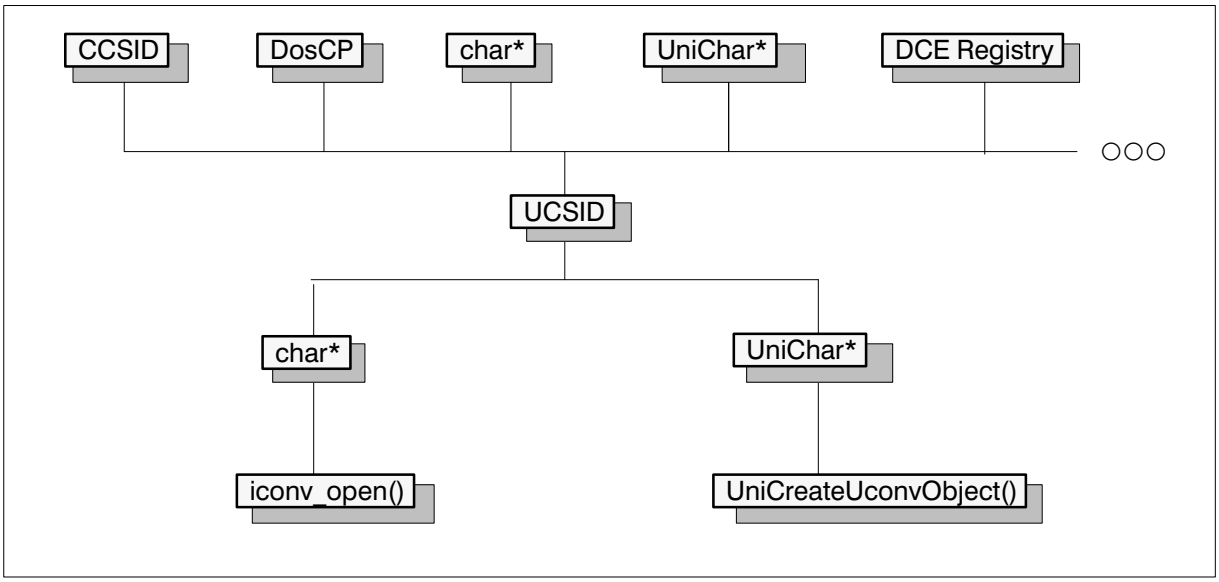


Figure 8. UCSID – Code Set Identifier Formats

9.5.1 List of Supported Tag Types

The following is a list of the types of tags that is supported by both **UniUcsidToTag**() and **UniUcsidFromTag**().

"CCSID" Specifies a tag of type CCSID (16 bits). The values of CCSID are defined by the IBM CDRA Registry. The *tag value* should be of type ***short****.

"DosCP" Specifies a tag in the format of a DOS Code Page. The *tag value* should be of type ***ulong****. It assumed that these tags values will be different then **ucsid**'s. If not, we should just use **ucsid** as is.

"CharString" Specifies a tag of type char*. The name should be limited to characters defined by the Portable Character Set.

”UCSString”	Specifies a tag of type ***UniChar****. The name should be limited to characters defined by the Portable Character Set.

9.5.2 UniUcsidToTag() – Map UCSID to identifier

```
#include <uconv.h>
int UniUcsidToTag(
        UCSID ucsid,                /* Specifies the UCSID to be mapped */
        UniChar *tagType,           /* Type of tag desired */
        void **tagValue             /* Tag to be returned */
        );
```

The **UniUcsidToTag**() function maps the **ucsid** into a **tagValue** as specified by **tagType.** The storage pointed to by the returned **tagValue** is owned by the **ULS** and should not be modified.

Return Value

If **ucsid** is a recognized **tagType**, a zero is returned and **tagValue** is set to point to tag value. Otherwise, **UniUcsidToTag**() returns the following value:

UCONV_EINVAL	The type of ***tag*** is not recognized by the implementation.

9.5.3 UniUcsidFromTag() – Map UCSID from identifier

#include <uconv.h>

int UniUcsidFromTag(

UCSID *ucsid,	**/* Specifies the UCSID to be returned */**
UniChar *tagType,	**/* Type of tag*/**
void *tagValue	**/* Tag to be converted into UCSID */**

The **UniUcsidFromTag**() function returns a **ucsid** based on the **tagValue** provided. The type and format of **tagValue** is specified by **tagType**.

Return Value

If both **tagType** and **tagValue** are recognized as valid tags, a zero is returned and **ucsid** is set with the corresponding **ucsid**. Otherwise, **UniUcsidFromTag**() returns the following value:

UCONV_EINVAL	The type of ***tag*** is not recognized by the implementation.

10 Commands and Utilities

This section describes system commands for creating locales, performing conversions and creating converters.

10.1 Locale management

10.1.1 localedef

localedef [−c] [−f *charmaplist*] [−i *sourcefilelist*] [−m *methodfile*] *localename*

The **localedef** utility converts source definitions for locale categories into a run−time format.

Options and operands:

−c Forces the creation of permanent output even if warning messages have been issued.

−f *charmaplist* Specifies a blank−separated list of pathnames for files containing mappings of character symbols and collating element symbols to actual character encodings. The list must be enclosed in quotes, so as to be interpreted as a single argument. These files are processed sequentially, with each subsequent file extending the definitions in previous files. In case of conflicts, previously listed files are given preference to later ones. The rules for this merging process and the format of these files are described in section 10.1.1.1.

−i *sourcefilelist* Specifies a blank−separated list of pathnames for files containing locale category source definitions. The list must be enclosed in quotes, so as to be interpreted as a single argument. These files are processed sequentially, with each subsequent file extending the definitions in previous files. In case of conflicts, previously listed files are given preference to later ones. The rules for this merging process and the format of these files are described in section 10.1.1.2. If this option is not present, source definitions are read from standard input.

−m *methodfile* The pathname of a file containing the mapping of method names to the functions and libraries implementing those methods. See section 10.1.1.3 for a specification of the format of this file.

localename The name of the locale to be created. Applications can use this name as an argument to the **UniCreateLocaleObject()** function to access the locale information.

10.1.1.1 Charmap Files

A charmap file is used to map symbolic character names to code points in a code set and to map characters to character sets.

The argument of the **−f** option of the **localedef** command is a list of one or more charmap files. These files are processed sequentially, with each subsequent file extending the definitions from previous files. In case of conflicts, previously listed files are given preference to later ones. The rules for this process are as follows:

1. In general, the overall definition is a union of all of the definitions in all of the listed charmap files.
2. If a new symbolic name is given for the same code point, the new name is treated as an alias for the previous name.
3. If a file attempts to map a previously defined symbolic name to a different code point, the new conflicting mapping is ignored.
4. If a file attempts to map a previously defined character to a new character set ID, the new conflicting mapping is ignored.
5. The list is always treated as if the **ISO 646** code set definition is at the beginning of the list. These characters are given standard symbolic names and assigned a charset id of **0**.

A charmap file contains a set of **declarations** followed by a **CHARMAP** section and a **CHARSET** section.

Declarations

The following declarations are permitted:

<escape_char>	The escape character used to indicate that the characters following will be interpreted in a special way, as defined later in this section. This defaults to backslash (\).
<code_set_name>	The name of the character encoding for which the charmap file is defined. The charmap file maps symbolic names to code points, the code set defines the mapping of code points to characters. The characters of the name must be taken from the the elements of the **portable character set** (Figure 9) with visible glyphs.
<mb_cur_max>	The maximum number of bytes in a multi−byte character. This defaults to 1.
<mb_cur_min>	The minimum number of bytes in a character.
<comment_char>	The character that is used to indicate that the remainder of a line is to be ignored. The default comment character is the number sign (**#**).

CHARMAP Section

The **CHARMAP** section maps symbolic character names to codepoints. In defining this mapping, one can assume that the mappings listed in Figure 9 are predefined.

CHARSET Section

The **CHARSET** section maps symbolic character names to numeric character set identifiers. These **CSID**s can be used to access the appropriate fonts used to display the characters.

Charmap Grammar

The following lexical conventions are used:

CHAR	Any single character
NUMBER	A decimal number, represented by one or more decimal digits

CHARSYMBOL	A symbolic name, enclosed between angle brackets.
OCTAL_CHAR	One or more octal representations of the encoding of each byte in a single character. The octal representation consists of an escape character (normally a backslash) followed by two or more octal digits.
HEX_CHAR	One or more hexadecimal representations of the encoding of each byte in a single character. The hexadecimal representation consists of an escape character followed by the constant **x** and two or more hexadecimal digits.
DECIMAL_CHAR	One or more decimal representations of the encoding of each byte in a single character. The decimal representation consists of an escape character followed by a character **d** and two or more decimal digits.
CS_NAME	A string of characters representing the name of the code set.
ELLIPSIS	The string **...**
EOL	The line termination character newline.

The following grammar defines the syntax for the charmap file:

```
charmap_definition
        : declarations charmap_sect charset_sect
        ;

declarations
        : declarations meta_symbol_spec
        | meta_symbol_spec
        ;

meta_symbol_spec
        : '<code_set_name>' CS_NAME EOL
        | '<mb_cur_max>' NUMBER EOL
        | '<mb_cur_min>' NUMBER EOL
        | '<escape_char>' CHAR EOL
        | '<comment_char>' CHAR EOL
        ;

charmap_sect
        : 'BEGIN' 'CHARMAP' EOL codeset_desc 'END' 'CHARMAP' EOL
        ;

codeset_desc
        : codeset_desc char_sym_enc
        | char_sym_enc
        ;

char_sym_range
        : char_sym_range encoding EOL
        ;

char_sym_range
        : CHARSYMBOL ELLIPSIS CHARSYMBOL
        | CHARSYMBOL
        ;
```

```
encoding
        : OCTAL_CHAR
        | HEX_CHAR
        | DEICMAL_CHAR
        ;

charset_sect
        : CHARSET cs_spec_list END CHARSET
        ;

cs_spec_list
        : cs_spec_list charset_spec
        | charset_spec
        ;

charset_spec
        : char_sym_range char_set_id EOL
        ;

char_set_id
        : number
        ;
```

Symbol Name	Code	Symbol Name	Code	Symbol Name	Code
<NUL>	000	<zero>	030	<grave−accent>	060
<SOH>	001	<one>	031	<a>	061
<STX>	002	<two>	032	<b>	062
<ETX>	003	<three>	033	<c>	063
<EOT>	004	<four>	034	<d>	064
<ENQ>	005	<five>	035	<e>	065
<ACK>	006	<six>	036	<f>	066
<alert>	007	<seven>	037	<g>	067
<backspace>	008	<eight>	038	<h>	068
<tab>	009	<nine>	039	<i>	069
<newline>	00A	<colon>	03A	<j>	06A
<vertical−tab>	00B	<semi−colon>	03B	<k>	06B
<form−feed>	00C	<less−than>	03C	<l>	06C
<carriage−return>	00D	<equal−sign>	03D	<m>	06D
<SO>	00E	<greater−than>	03E	<n>	06E
<SI>	00F	<question−mark>	03F	<o>	06F
<DLE>	010	<commercial−at>	040	<p>	070
<DC1>	011	<A>	041	<q>	071
<DC2>	012	<B>	042	<r>	072
<DC3>	013	<C>	043	<s>	073
<DC4>	014	<D>	044	<t>	074
<NAK>	015	<E>	045	<u>	075
<SYN>	016	<F>	046	<v>	076
<ETB>	017	<G>	047	<w>	077
<CAN>	018	<H>	048	<x>	078
<EM>	019	<I>	049	<y>	079
<SUB>	01A	<J>	04A	<z>	07A
<ESC>	01B	<K>	04B	<left−brace>	07B
<IS4>	01C	<L>	04C	<vertical−line>	07C
<IS3>	01D	<M>	04D	<right−brace>	07D
<IS2>	01E	<N>	04E	<tilde>	07E
<IS1>	01F	<O>	04F	<DEL>	07F
<space>	020	<P>	050		
<exclamation−mark>	021	<Q>	051		
<quotation−mark>	022	<R>	052		
<number−sign>	023	<S>	053		
<dollar−sign>	024	<T>	054		
<percent>	025	<U>	055		
<ampersand>	026	<V>	056		
<apostrophe>	027	<W>	057		
<left−parenthesis>	028	<X>	058		
<right−parenthesis>	029	<Y>	059		
<asterisk>	02A	<Z>	05A		
<plus−sign>	02B	<left−bracket>	05B		
<comma>	02C	<backslash>	05C		
<hyphen>	02D	<right−bracket>	05D		
<period>	02E	<circumflex>	05E		
<slash>	02F	<underscore>	05F		

Figure 9. Portable character set, symbolic names and code points

10.1.1.2 Locale Source File

A locale source file consists of a list of category definitions. Each category definition is identified by the corresponding environment variable name and terminated by an **END *category−name*** statement. The following categories are supported:

LC_CTYPE	Defines character classification and case conversion.
LC_COLLATE	Defines collation rules.
LC_MONETARY	Defines the format and symbols used in formatting of monetary information.
LC_NUMERIC	Defines the decimal delimiter, grouping and grouping symbol for non−monetary numeric editing.
LC_TIME	Defines the format and content of date and time information.
LC_MESSAGES	Defines the format and values of affirmative and negative responses.

The argument of the **−i** option of the **localedef** command is a list of one or more locale source files. These files are processed sequentially, with each subsequent file extending the definitions from the previous file. In case of conflicts, previously listed files are given preference to later ones. The rules for this process are as follows:

1. The list is always treated as if the **C** locale definition is at the end of the list. If something from the **C** locale definition is not defined in any of the listed files, the **C** locale definition will be used.
2. The definitions for the **LC_MONETARY**, **LC_NUMERIC**, **LC_TIME** and **LC_MESSAGES** categories are extended on a line by line (keyword by keyword) basis. If a file attempts to redefine a keyword which was defined in a previous file, the new conflicting definition is ignored. In general, a token specified in a previous file completely overrides definition for that token in later files.
3. For the **LC_CTYPE** category, each keyword defines a list of characters or character transformations. In this case, new definitions are permitted to extend previous definitions, and the overall definition will be the union of all definitions in all files.

   Conflicts between definitions in different files are resolved by using the earlier definition and ignoring the later conflicting one.

   Conflicts arise because some of the standard attributes are defined to be mutually exclusive, as described in the table in Figure 10 and because of the **toupper** and **tolower** mappings. If a file attempts to classify a character in a way which would cause it to belong to mutually exclusive classes, either explicitly or implicitly (by automatic inclusion as defined in Figure 10), this new conflicting definition is ignored.

   The character classes **digit**, **xdigit**, **lower**, **upper** and **space** have required sets of automatically included characters defined for them. The **localedef** command behaves as if these **LC_CTYPE** definitions were defined before all the definitions in the user−specified list of source files.

   For the. **toupper** and **tolower** mappings, the final definition is the union of all definitions from all files. In case of conflicts, the earlier mapping for a particular character is used. Within a single file, if a **toupper** definition exists, but a **tolower** definition is absent, that file will be treated as if both definitions were present, and the **toupper** is the reverse of the **tolower**.
4. For the **LC_COLLATE** category, all definitions are merged in such a way that no conflicts exist. The overall collation specification is obtained by imposing all of the collation rules in the individual files

and giving collation elements in earlier files weights lower than those in later files. Collation element definitions and collation orderings which conflict with those in earlier files are ignored.

LC_CTYPE

The **LC_CTYPE** category defines character classification, case conversion and other character attributes. The following keywords are recognized:

upper Defines characters to be classified as upper−case letters. The upper−case letters **A** to **Z** from the portable character set are automatically included in this class.

lower Defines characters to be classified as lower−case letters. The upper−case letters **a** to **z** from the portable character set are automatically included in this class.

alpha Defines characters to be classified as letters.

digit Defines characters to be classified as numeric digits. These are the digits **0** to **9** of the portable character set.

space Defines characters to be classified as white−space characters. The characters **space**, **form−feed**, **newline**, **carriage−return**, **tab** and **vertical−tab** from the portable character set are automatically included in this class.

cntrl Defines characters to be classified as control characters.

punct Defines characters to be classified as punctuation characters.

graph Defines characters to be classified as printable characters, not including the space character.

print Defines characters to be classified as printable characters, including the space character.

xdigit Defines characters to be classified as hexadecimal digits. The digits **0** to **9**, the upper−case letters **A** to **F** and the lower−case letters **a** to **f** of the portable character set are automatically included in this class.

blank Defines the characters to be classified as blank characters. The characters **space** and **tab** are automatically included in this class.

copy Specifies the name of an existing locale to be used as the definition of this category.

A series of characters can also be represented by three adjacent periods representing an ellipsis symbol (**...**). An ellipsis is interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

The table in Figure 10 shows the valid combinations for the predefined properties.

In	Can Also Belong To										
Class	upper	lower	alpha	digit	space	cntrl	punct	graph	print	xdigit	blank
upper		–	A	X	X	X	X	A	A	–	X
lower	–	–	A	X	X	X	X	A	A	–	X
alpha	–	–		X	X	X	X	A	A	–	X
digit	X	X	X		X	X	X	A	A	A	X
space	X	X	X	X		–	*	*	*	X	–
cmtrl	X	X	X	X	–		X	X	X	X	–
punct	X	X	X	X	–	X		A	A	X	–
graph	–	–	–	–	–	X	–		A	–	–
print	–	–	–	–	–	X	–	–		–	–
xdigit	–	–	–	–	X	X	X	A	A		X
blank	X	X	X	X	A	–	*	*	*	X	

Figure 10. Valid Character Class Combinations

Legend

A	Automatically included
–	Permitted
X	Mutually exclusive
*	The **<space>** character, which has the **space** and **blank** properties cannot have the **punct** or **graph** properties, but will automatically have the **print** property. Other characters with the **space** or **blank** property may have the **punct**, **graph** or **print** property.

LC_COLLATE

The **LC_COLLATE** category provides a collation sequence definition, which defines the relative order between collating elements (characters and multi−character collating elements) in the locale. This order is expressed in terms of collation values' that is, by assigning each element one or more collation values (also known as collation weights). This does not imply that implementations assign such values, but that ordering of strings using the resultant collation definition in the locale will behave as if such assignment is done and used in the collation process. At least the following capabilities are provided:

1. Multi−character collating elements.
2. User−defined ordering of collating elements.
3. Multiple weights and equivalence classes.
4. One−to−Many mapping.
5. Equivalence class definition.
6. Ordering by weights.

The following keywords are recognized in a collation sequence definition:

collating−element	Define a collating−element symbol representing a multi−character collating element.
collating−symbol	Define a collating symbol for use in collation order statements.

order_start	Define collation rules. This statement is followed by one or more collation order statements, assigning character collation values and collation weights to collating elements.
order_end	Specify the end of the collation−order statements.
copy	Specify the name of an existing locale to be used as the definition of this category. If this keyword is specified, no other keyword can be specified.

LC_MONETARY
The **LC_MONETARY** category defines the rules and symbols that are used to format monetary numeric information.

LC_NUMERIC
The **LC_NUMERIC** category defines the rules and symbols that will be used to format non−monetary numeric information.

LC_TIME
The **LC_TIME** category defines the rules and symbols that will be used to format dates and times.

LC_MESSAGES
The **LC_MESSAGES** category defines the format and values for affirmative and negative responses.

For a complete description of the syntax for the locale source file, refer to the X/Open specification ([27]).

Locale Source File Grammar

The following lexical conventions are used:

LOC_NAME	A string of characters representing the name of a locale
CHAR	Any single character
NUMBER	A decimal number, represented by one or more decimal digits
COLLSYMBOL	A symbolic name, enclosed between angle brackets, The string cannot duplicate any charmap symbol defined in the current charmap (if any), or a COLLELEMENT symbol.
COLLELEMENT	A symbolic name, enclosed between angle brackets, which cannot duplicate either any charmap symbol or a COLLSYMBOL symbol.
CHARSYMBOL	A symbolic name, enclosed between angle brackets.
CHARCLASS	A string of alphanumeric characters from the portable characters set, the first of which is not a digit, consisting of at least one and at most {CHARCLASS_NAME_MAX} bytes, and optionally surrounded by double−quotes.
OCTAL_CHAR	One or more octal representations of the encoding of each byte in a single character. The octal representation consists of an escape character (normally a backslash) followed by two or more octal digits.
HEX_CHAR	One or more hexadecimal representations of the encoding of each byte in a single character. The hexadecimal representation consists of an escape character followed by the constant **x** and two or more hexadecimal digits.

DECIMAL_CHAR — One or more decimal representations of the encoding of each byte in a single character. The decimal representation consists of an escape character followed by a character **d** and two or more decimal digits.

ELLIPSIS — The string **...**

EXTENDED_REG_EXP — An extended regular expression.

EOL — The line termination character newline.

The following grammar defines the syntax for the charmap file:

```
locale_definition
        : global_statements locale_categories
        | locale_categories
        ;

global_statements
        : global_statements symbol_redefine
        | symbol_redefine
        ;

symbol_redefine
        : 'escape_char' CHAR EOL
        | 'comment_char' CHAR EOL
        ;

locale_categories
        : locale_categories locale_category
        | locale_category
        ;

locale_category
        : lc_ctype
        | lc_collate
        | lc_messages
        | lc_monetary
        | lc_numeric
        | lc_time
        ;
```

The following grammar rules are common to all categories:

```
char_list
        : char_list char_symbol
        | char_symbol
        ;

char_symbol
        : CHAR
        | CHARSYMBOL
        | OCTAL_CHAR
        | HEX_CHAR
        | DECIMAL_CHAR
        ;
```

```
elem_list
        : elem_list char_symbol
        | elem_list COLLSYMBOL
        | elem_list COLLELEMENT
        | char_symbol
        | COLLSYMBOL
        | COLLELEMENT
        ;


symb_list
        : symb_list COLLSYMBOL
        | COLLSYMBOL
        ;

locale_name
        : LOC_NAME
        | '"' LOC_NAME '"'
        ;
```

The following is the **LC_CTYPE** category grammar:

```
lc_ctype
        : ctype_hdr ctype_keywords ctype_tlr
        | ctype_hdr 'copy' locale_name EOL ctype_tlr
        ;

ctype_hdr
        : 'LC_CTYPE' EOL
        ;

ctype_keywords
        : ctype_keywords ctype_keyword
        | ctype_keyword
        ;

ctype_keyword
        : charclass_keyword charclass_list EOL
        | charconv_keyword charconv_list EOL
        | 'charclass' charclass_namelist EOL
        ;

charclass_namelist
        : charclass_namelist ';' CHARCLASS
        | CHARCLASS
        ;

charclass_keyword
        : 'upper'
        | 'lower'
        | 'alpha'
        | 'digit'
        | 'punct'
        | 'xdigit'
        | 'space'
        | 'print'
        | 'graph'
        | 'blank'
        | 'cntrl'
        | CHARCLASS
        ;
```

```
charclass_list
        : charclass_list ';' char_symbol
        | charclass_list ';' ELLIPSIS ';' char_symbol
        | char_symbol
        ;

charconv_keyword
        : 'toupper'
        | 'tolower'
        | 'symmetric'
        ;

charconv_list
        : charconv_list ';' charconv_entry
        | charconv_entry
        ;

charconv_entry
        : '(' char_symbol ',' char_symbol ')'
        ;

ctype_tlr
        : 'END' 'LC_CTYPE' EOL
        ;
```

The following is the **LC_COLLATE** category grammar:

```
lc_collate
        : collate_hdr collate_keywords collate_tlr
        | collate_hdr 'copy' locale_name EOL collate_tlr
        ;

collate_hdr
        : 'LC_COLLATE' EOL
        ;

collate_keywords
        : order_statements
        | opt_statements order_statements
        ;

opt_statements
        : opt_statements collating_symbols
        | opt_statements collating_elements
        | collating_symbols
        | collating_elements
        ;

collating_symbols
        : 'collating-symbol' COLLSYMBOL EOL
        ;

collating_elements
        : 'collating-element' COLLELEMENT
          'from' '"' elem_list '"' EOL
        ;

order_statements
        : order_start collation_order order_end
        ;

order_start
        : 'order_start' EOL
        | 'order_start' order_opts EOL
        ;
```

```
order_opts
        : order_opts ';' order_opt
        | order_opt
        ;


order_opt
        : order_opt ',' opt_word
        | opt_word
        ;
opt_word
        : 'forward'
        | 'backward'
        | 'position'
        ;
collation_order
        : collation_order collation_entry
        | collation_entry
        ;
collation_entry
        : COLLSYMBOL EOL
        | collation_element weight_list EOL
        | collation_element EOL
        ;
collation_element
        : char_symbol
        | COLLELEMENT
        | ELLIPSIS
        | 'UNDEFINED'
        ;
weight_list
        : weight_list ';' weight_symbol
        | weight_list ';'
        | weight_symbol
        ;
weight_symbol
        | char_symbol
        | COLLSYMBOL
        | '"' elem_list '"'
        | '"' symb_list '"'
        | ELLIPSIS
        | 'IGNORE'
        : /* empty */
        ;
order_end
        : 'order_end' EOL
        ;
collate_tlr
        : 'END' 'LC_COLLATE' EOL
        ;
```

The following is the **LC_MESSAGES** category grammar:

```
lc_messages
        : messages_hdr messages_keywords messages_tlr
        | messages_hdr 'copy' locale_name EOL messages_tlr
        ;

messages_hdr
        : 'LC_MESSAGES' EOL
        ;

messages_keywords
        : messages_keywords messages_keyword
        | messages_keyword
        ;

messages_keyword
        : 'yesexpr' '"' EXTENDED_REG_EXP '"' EOL
        | 'noexpr' '"' EXTENDED_REG_EXP '"' EOL
        | 'yesstr' '"' char_list '"' EOL
        | 'nostr' '"' char_list '"' EOL
        ;

messages_tlr
        : 'END' 'LC_MESSAGES' EOL
        ;
```

The following is the **LC_MONETARY** category grammar:

```
lc_monetary
        : monetary_hdr monetary_keywords monetary_tlr
        | monetary_hdr 'copy' locale_name EOL monetary_tlr
        ;

monetary_hdr
        : 'LC_MONETARY' EOL
        ;

monetary_keywords
        : monetary_keywords monetary_keyword
        | monetary_keyword
        ;

monetary_keyword
        : mon_keyword_string mon_string EOL
        | mon_keyword_char NUMBER EOL
        | mon_keyword_char '-1' EOL
        | mon_keyword_grouping mon_group_list EOL
        ;

mon_keyword_string
        : 'int_curr_symbol'
        | 'currency_symbol'
        | 'mon_decimal_point'
        | 'mon_thousands_sep'
        | 'positive_sign'
        | 'negative_sign'
        ;

mon_string
        : '"' char_list '"'
        | '""'
        ;
```

```
mon_keyword_char
        : 'int_frac_digits'
        | 'frac_digits'
        | 'p_cs_precedes'
        | 'p_sep_by_space'
        | 'n_cs_precedes'
        | 'n_sep_by_space'
        | 'p_sign_posn'
        | 'n_sign_posn'
        ;

mon_keyword_grouping
        : 'mon_grouping'
        ;

mon_group_list
        : NUMBER
        | mon_group_list ';' NUMBER
        ;

monetary_tlr
        : 'END' 'LC_MONETARY' EOL
        ;
```

The following is the **LC_NUMERIC** category grammar:

```
lc_numeric
        : numeric_hdr numeric_keywords numeric_tlr
        | numeric_hdr 'copy' locale_name EOL numeric_tlr
        ;

numeric_hdr
        : 'LC_NUMERIC' EOL
        ;

numeric_keywords
        : numeric_keywords numeric_keyword
        | numeric_keyword
        ;

numeric_keyword
        : num_keyword_string num_string EOL
        | num_keyword_grouping num_group_list EOL
        ;

num_keyword_string
        : 'decimal_point'
        | 'thousands_sep'
        ;

num_string
        : '"' char_list '"'
        | '""'
        ;

num_keyword_grouping
        : 'grouping'
        ;

num_group_list
        : NUMBER
        | num_group_list ';' NUMBER
        ;
```

```
numeric_tlr
        : 'END' 'LC_NUMERIC' EOL
        ;
```

The following is the **LC_TIME** category grammar:

```
lc_time
        : time_hdr time_keywords time_tlr
        | time_hdr 'copy' locale_name EOL time_tlr

time_hdr
        : 'LC_TIME' EOL
        ;

time_keywords
        : time_keywords time_keyword
        | time_keyword
        ;

time_keyword
        : time_keyword_name time_list EOL
        | time_keyword_fmt time_string EOL
        | time_keyword_opt time_list EOL
        ;

time_keyword_name
        : 'abday'
        | 'day'
        | 'admon'
        | 'mon'
        ;

time_keyword_fmt
        : 'd_t_fmt'
        | 'd_fmt'
        | 't_fmt'
        | 'am_pm'
        | 't_fmt_ampm'
        ;

time_keyword_opt
        : 'era'
        | 'era_d_fmt'
        | 'era_t_fmt'
        | 'era_d_t_fmt'
        | 'alt_digits'
        ;

time_list
        : time_list ';' time_string
        | time_string
        ;

time_string
        : '"' char_list '"'
        ;

time_tlr
        : 'END' 'LC_TIME' EOL
        ;
```

10.1.1.3 Methods File

The methods file maps method names to the names of the **C** functions which implement those methods and specifies the libraries in which the implementing functions may be found. It also provides information which

the localedef utility uses to construct any tables or other supporting framework which the methods might depend on.

The methods correspond to those functions which require direct access to the data structures representing locale data. The syntax for specifying these methods follows:

Methods File Grammar

The following lexical conventions are used:

FUNC_NAME	A string of characters representing the name of a function.
PATH_NAME	A POSIX portable path name.
EOL	The line termination character newline.

The following grammar defines the syntax for the methods file:

```
methods_definition
        : framework_spec method_sect
        ;

framework_spec
        TBD
        ;

method_sect
        : 'METHODS' EOL method_spec 'END' 'METHODS' EOL
        ;

method_spec
        : method_spec method_stat
        | method_stat
        ;


method_stat
        : method_name funct_name lib_name EOL
        | method_name funct_name EOL
        ;
```

```
    method_name
                                    # XPG I18N (UTF/UCS) Methods
            : 'mblen'
            | 'mbtowc'
            | 'mbstowcs'
            | 'wctomb'
            | 'wcstombs'
            | 'wcwidth'
            | 'wcswidth'
            | 'wcstod'
            | 'wcstol'
            | 'wcstoul'
            | '__mbtopc'
            | '__mbstopcs'
            | '__pctomb'
            | '__pcstombs'
            | 'csid'
            | 'wcsid'
            | 'towupper'
            | 'towlower'
            | 'get_wctype'
            | 'is_wctype'
            | 'strcoll'
            | 'strxfrm'
            | 'wcscoll'
            | 'wcsxfrm'
            | 'regcomp'
            | 'regexec'
            | 'regfree'
            | 'regerror'
            | 'strfmon'
            | 'strftime'
            | 'strptime'
            | 'wcsftime'
                                    # Universal Locale Methods
            | 'unicreatelocale'
            | 'unifreelocale'
            | 'unicsid'
            | 'unicoll'
            | 'unixfrm'
            | 'uniftime'
            | 'uniwidth'
            | 'unitod'
            | 'unitol'
            | 'unitoul'
            | 'unicreateattr'
            | 'uniscanforattr'
            | 'unifreeattr'
            | 'unicreatetransform'
            | 'unitransform'
            | 'unifreetransform'
                                    # Universal IO Methods
            | '__unifrommbs'
            | '__unitombs'
            ;
```

For implementations where **wchar_t** and **UniChar** are the same size (16 bits) some of the methods will be shared –

unicsid = wcsid

```
unicoll         = wcscoll
unixfrm         = wcsxfrm
uniftime        = wcsftime
uniwidth        = wcswidth
unitod          = wcstod
unitol          = wcstol
unitoul         = wcstoul
__unifrommbs    = __mbstopcs
__unitombs      = __pcstombs

  funct_name
          : '"' FUNCT_NAME '"'
          ;

  lib_name
          : '"' PATH_NAME '"'
          ;
```

The **localedef** command will parse this information to determine the methods to be used for this locale. The following 10 methods **must** be specified in the method file, if it exists.

__mbstopcs
__mbtopc
__pcstombs
__pctomb
mblen
mbstowcs
mbtowc
wcstombs
wcswidth
wctomb
wcwidth

Any other method not specified in the method file will use the corresponding default method.

Note that in the syntax for the **METHODS** file, it is not mandatory to specify the library name. If the library name is not specified, the method will inherit the most recently specified library, or if no library has been specified, the library **libc.a**.

10.2 Layout management

10.2.1 layoutdef

layoutdef [−c] [−f *charmaplist*] [−i *sourcefilelist*] [−m *methodfile*] *layoutname*

The **layoutdef** utility converts source definitions for layout categories into a run−time format.

Options and operands:

−c Forces the creation of permanent output even if warning messages have been issued.

−f *charmaplist* Specifies a blank−separated list of pathnames for files containing mappings of character symbols and collating element symbols to actual character encodings. The list must be enclosed in quotes, so as to be interpreted as a single argument. These files are processed sequentially, with each subsequent file extending the definitions in previous files. In case of conflicts, previously listed files are given preference to later ones. The rules for this merging process and the format of these files are described in section 10.1.1.1.

−i *sourcefilelist* Specifies a blank−separated list of pathnames for files containing layout category source definitions. The list must be enclosed in quotes, so as to be interpreted as a single argument. These files are processed sequentially, with each subsequent file extending the definitions in previous files. In case of conflicts, previously listed files are given preference to later ones. The rules for this merging process and the format of these files are described in section 10.2.1.1. If this option is not present, source definitions are read from standard input.

−m *layoutmethodfile* The pathname of a file containing the mapping of method names to the functions and libraries implementing those methods. See section 10.2.1.2 for a specification of the format of this file.

layoutname The name of the layout to be created. Applications can use this name as an argument to the **UniCreateLayoutObject**() function to access the layout information.

10.2.1.1 Layout Source File

A layout source file consists of a list of category definitions. Each category definition is identified by the corresponding environment variable name and terminated by an **END *category−name*** statement. The following layout categories are supported:

LC_LTYPE Defines per character layout classification and layout conversion.

The argument of the **−i** option of the **layoutdef** command is a list of one or more layout source files. These files are processed sequentially, with each subsequent file extending the definitions from the previous file. In case of conflicts, previously listed files are given preference to later ones. The rules for this process are as follows:

1. The list is always treated as if the **C** layout definition is at the end of the list. If something from the **C** layout definition is not defined in any of the listed files, the **C** layout definition will be used.
2. For the **LC_LTYPE** category, each keyword defines a list of characters or character transformations. In this case, new definitions are permitted to extend previous definitions, and the overall definition will be the union of all definitions in all files.

   Conflicts between definitions in different files are resolved by using the earlier definition and ignoring the later conflicting one.

Conflicts arise because some of the standard attributes are defined to be mutually exclusive, as described in the table in Figure 10 and because of the **toupper** and **tolower** mappings. If a file attempts to classify a character in a way which would cause it to belong to mutually exclusive classes, either explicitly or implicitly (by automatic inclusion as defined in Figure 10), this new conflicting definition is ignored.

The character classes **RIGHT_TO_LEFT**, **LEFT_TO_RIGHT**, **NUM_TERMINATOR**, **NATIONAL_NUMBER**, **COMMON_NUM_SEPARATOR, SEGMENT_SEPERATOR,** and **BLOCK_SEPERATOR** have required sets of automatically included characters defined for them. The **layoutdef** command behaves as if these **LC_CTYPE** definitions were defined before all the definitions in the user−specified list of source files.

For any of the mappings (**TOSYMMETRIC**, **TONATIONAL**, **TODIGIT**) the final definition is the union of all definitions from all files. In case of conflicts, the earlier mapping for a particular character is used

LC_LTYPE
The **LC_LTYPE** category defines the text attributes related to directionality,shaping and numeric selection. The following keywords are recognized:

LEFT_TO_RIGHT Defines characters to be classified as characters with Left−To−Right directionality.

RIGHT_TO_LEFT Defines characters to be classified as characters with Right−To−Left directionality.

NUM_SEPARATOR Defines characters to be classified as separators of numerals of the portable digit set (but not of national numerals).

NUM_TERMINATOR Defines characters to be classified as terminators of numerals.

NATIONAL_NUMBER Defines characters to be classified as national numbers (For example Hindi numerals used in Arabic countries in Arabic script, Thai numerals used in Thai script, Chinese numerals used in Chinese vertical script, Bengali numerals used in Bengali script).

COMMON_NUM_SEPARATOR Defines characters to be classified as numbers separators both for the numerals of the portable character set and for national numerals.

SEGMENT_SEPARATOR Defines segment separator characters. A segment is a portion of text, in general shorter than one line, imbedded within a wider text that has a different directionality.

BLOCK_SEPARATOR Defines block separator characters. A block is a larger part of text (one or more paragraphs) with a distinct directionality that may differ from the directionality of other parts of text in a document.

DIRECTION_CONTROL Defines characters to be classified as directional control characters. Direction control characters are described in ISO/IEC 10646−1 1993 Annex D.

SYM_SWAP_FORMAT Defines characters to be classified as symmetrical swap format characters. Symmetrical swap format characters are described in ISO/IEC 10646−1 1993 Annex D.

CHAR_SHAPE_SELECTOR Defines characters to be classified as character shaping selectors characters. Character shaping selectors are described in ISO/IEC 10646−1 1993 Annex D.

NUM_SHAPE_SELECTOR Defines characters to be classified as numeric shaping selectors characters. Numeric shaping selectors are described in ISO/IEC 10646−1 1993 Annex D.

NATIONAL_CHAR Defines characters to be classified as national characters.

TONATIONAL Defines the mapping of digits to their corresponding national digits. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair is surrounded by parentheses. The first character in the pair represents a nominal digit, while the second represents a national digit. A nominal digit is one that belongs to the set of digits in the portable character set.

TODIGIT Defines the mapping of national digits to their corresponding digits. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair is surrounded by parentheses. The first character in the pair represents a national digit, while the second represents a nominal digit.

TOSYMMETRIC Defines the mapping of symmetric characters that can be swapped. The operand consists of character pairs, separated by semicolons. The characters in each character pair are separated by a comma and the pair surrounded by parentheses. The first character of each pair is to be swapped with the second one. Symmetric characters are defined in ISO/IEC 10646−1:1993(E), clause 20.

copy Specifies the name of an existing layout to be used as the definition of this category.

A series of characters can also be represented by three adjacent periods representing an ellipsis symbol (**...**). An ellipsis is interpreted as including in the list all characters with an encoded value higher than the encoded value of the character preceding the ellipsis and lower than the encoded value of the character following the ellipsis.

Layout Source File Grammar

The lexical conventions used within the Layout Source file are the same as those defined for the locale source file grammer, refer to **Locale Source File Grammer** in 10.1.1.2.

The layout source The following is the **LC_LTYPE** category grammar: (This grammer

```
LC_LTYPE
        : layout_hdr layout_keywords layout_tlr
        | layout_hdr 'copy' locale_name EOL layout_tlr
        ;
```

```
layout_hdr
        : 'LC_LTYPE' EOL
        ;
layout_keywords
        : layout_keywords layout_keyword
        | layout_keyword
        ;
layout_keyword
        : charclass_keyword charclass_list EOL
        | charconv_keyword charconv_list EOL
        | 'charclass' charclass_namelist EOL
        ;
charclass_namelist
        : charclass_namelist ';' CHARCLASS
        | CHARCLASS
        ;
charclass_keyword
        : 'LEFT_TO_RIGHT'
        | 'RIGHT_TO_LEFT'
        | 'NUM_SEPARATOR'
        | 'NUM_TERMINATOR'
        | 'NATIONAL_NUMBER'
        | 'COMMON_NUM_SEPARATOR'
        | 'SEGMENT_SEPARATOR'
        | 'BLOCK_SEPARATOR'
        | 'DIRECTION_CONTROL'
        | 'SYM_SWAP_FORMAT'
        | 'CHAR_SHAPE_SELECTOR'
        | 'NUM_SHAPE_SELECTOR'
        | 'NATIONAL_CHAR'
        ;
charclass_list
        : charclass_list ';' char_symbol
        | charclass_list ';' ELLIPSIS ';' char_symbol
        | char_symbol
        ;
charconv_keyword
        : 'TONATIONAL'
        | 'TODIGIT'
        | 'TOSYMMETRIC'
        ;
charconv_list
        : charconv_list ';' charconv_entry
        | charconv_entry
        ;
charconv_entry
        : '(' char_symbol ',' char_symbol ')'
        ;
layout_tlr
        : 'END' 'LC_LTYPE' EOL
        ;
```

10.2.1.2 LayoutMethods File

The layout methods file maps method names to the names of the **C** functions which implement those layout methods and specifies the libraries in which the implementing functions may be found. It also provides information which the layoutdef utility uses to construct any tables or other supporting framework which the methods might depend on.

The methods correspond to those functions which require direct access to the data structures representing layout data. The syntax for specifying these methods follows:

Layout Methods File Grammar

The following lexical conventions are used:

FUNC_NAME	A string of characters representing the name of a function.
PATH_NAME	A POSIX portable path name.
EOL	The line termination character newline.

The following grammar defines the syntax for the methods file:

```
methods_definition
        : framework_spec method_sect
        ;

framework_spec
        TBD
        ;

method_sect
        : 'METHODS' EOL method_spec 'END' 'METHODS' EOL
        ;

method_spec
        : method_spec method_stat
        | method_stat
        ;


method_stat
        : method_name funct_name lib_name EOL
        | method_name funct_name EOL
        ;

method_name
        : 'transform'
        | 'setvalues'
        | 'getvalues'
        | 'editshape'
        | 'createattr'
        | 'scanfor';

funct_name
        : '"' FUNCT_NAME '"'
        ;

lib_name
        : '"' PATH_NAME '"'
        ;
```

The **layoutdef** command will parse this information to determine the methods to be used for this layout. The following layout methods **must** be specified in the method file, if it exists.

uniCreateLayout
uniFreeLayout
uniTransformLayout
uniEditLayoutShape
uniSetLayoutValues
uniGetLayoutValues
uniCreateLayoutAttr
uniScanforLayoutAttr
uniFreeLayoutAttr

Any other method not specified in the method file will use the corresponding default method.

Note that in the syntax for the **METHODS** file, it is not mandatory to specify the library name. If the library name is not specified, the method will inherit the most recently specified library, or if no library has been specified, the library **libc.a**.

10.3 Conversion

10.3.1 iconv Command

iconv −f *fromcode* −t *tocode* [*filename*]

The **iconv** command converts characters from **stdin** or ***filename***, and sends the converted data to **stdou**t. The input data is expected to consist of characters in the ***fromcode*** codeset. This data is converted to data in the ***tocode*** codeset and sent to **stdout**.

Options and operands:

−f *fromcode*	Specifies the codeset in which the input data is encoded.
−t *tocode*	Specifies the codeset in which the output data is to be converted.
filename	Pathname of the input file to be converted. If *filename* is omitted, standard input is used.

Exit Status:

0	Successful completion
>0	An error occurred

10.3.2 uconvdef Command

uconvdef [−f *srcfile*] [−m] [−v] *uconvtable*

The uconvdef command reads the srcfile source definition file and creates a compiled version of the file which can be used by the **iconv** command and the **iconv** and **uconv** programming interfaces. The source code set conversion table file contains mappings between **UCS−2** and another codeset and directives that are acted upon by the **uconvdef** command to produce the compiled version.

Options and operands:

−f *srcfile*	Specifies the pathname of the **uconvdef source file** which contains the conversion mapping and parameters. If this option is not given, the specification is read from standard input. The format of this file is described in section 10.3.2.1.
−v	Verbose flag, which causes the program to output the source file statements processed.
uconvtable	Pathname of the compiled conversion table file to be generated. In general, this should be the name of the code set which is being converted into and out of UCS.

Exit Status:

0	Successful completion
>0	An error occurred

10.3.2.1 Uconv Source File

Coded character set character values are defined using symbolic character names followed by character encoding (code point) values, such as, **<U0020> \x20**, which represents the space in ASCII.

The following declarations can precede the character definitions. Each declaration consists of the symbol shown in the following list, starting in column 1, including the surrounding brackets, followed by one or more blanks, followed by the value to be assigned to the symbol:

<code_set_name>	The name of the coded character set for which the character set description file is defined. The characters of the name must be taken from the the elements of the **portable character set** (Figure 9) with visible glyphs, and are enclosed in quotation marks.
<mb_cur_max>	The maximum number of bytes in a multibyte character. This defaults to 1.
<mb_cur_min>	An unsigned positive integer value that defines the minimum number of bytes in a character for the encoded character set. The value is less than or equal to **<mb_cur_max>**. If not specified, the minimum number will be equal to **<mb_cur_max>**.
<escape_char>	The escape character used to indicate that the characters following is interpreted in a special way, as defined later in this section. This defaults to backslash (\), which is the character glyph used in all the following text examples .
<comment_char>	The character, that when placed in column 1 of a charmap line, is used to indicate that the line is ignored. The default character will be the number−sign (#).
<char_name_mask>	A quoted string consisting of format specifiers, which are represented by one of the following characters: **A**, **X**, **O**, and **D**. **A** indicates an alphabetic character from the set shown with visible glyphs. **X** indicates a hexadecimal digit, **O** indicates an

octal digit, and **D** indicates a decimal digit. One or more **A**s will be followed by any number of **X**s, or **O**s, or **D**s. The mask specifies the structure of the symbolic character names used in the coded character set description file, but it does not apply to the reserved symbolic character name **<unassigned>**described later in this section.

The ULS is based on a universal code set model, so all characters are expressed in terms of code points of the universal code set (UCS−2) For this reason the character name mask **"AXXXX"** is the only one supported in the ULS. The alphabetic character in the mask is set to **U**, while the 4 hexadecimal digits represent a UCS−2 code point. An example of a symbolic character name based on this mask is **<U0020>** which is the Unicode space character.

<locale> A reserved symbolic character name indicating the locale to be use if locale information is needed.

<uconv_class> The class of conversion that this table will be associated with. Each implementation will define a set of classes that it supports. The following set of **uconv_class** will be support all implementations of **ULS:**

"SBCS"	– single byte encodings
"MBCS"	– generic multibyte encodings
"BIDI"	– Bidirectional encodings.
"DBCS−PC"	– PC DBCS codeset
"DBCS−EBCDIC"	– EBCDIC with DBCS code sets

<subchar> A reserved symbolic character name indicating that a code point that indicates the default substitution character. This symbolic name can be specified more than once in the coded character set description file.

The character set mapping definitions are all the lines immediately following an identifier line containing the string **CHARMAP**, starting in column 1, and preceding a trailer line containing the string **END CHARMAP** starting in column 1. Empty lines and lines containing **<comment_char>** in the first column are ignored. Symbolic character names follow the pattern specified in the **<char_name_mask>**. This does not apply to the reserved symbolic character names described below.

<unassigned> A reserved symbolic character name indicating that a code point (or a range of code points) with which it is associated is unassigned. This symbolic name can be specified more than once in the coded character set description file.

In each noncomment line of the character set mapping definition (ie., between the **CHARMAP** and **END CHARMAP** lines of the file) a symbolic character name follows the structure defined by the value of the **<char_name_mask>**, and it is enclosed in angle brackets. This does not apply to the reserved symbolic character names. Each noncomment line of the character set mapping definition must be in one of the following forms:

1. **"%s %s %s\n"**, <*symbolic−name*>, <*encoding*>, <*comments*>

   Example:

```
<U3004>              \x81\x57
```

In this form the line in the character set mapping definition defines a single symbolic character name and a corresponding encoding.

A character following an escape character is interpreted as itself; for example, the sequence **<\\\>>** represents the symbolic name **\>** enclosed between angle brackets.

2. **"%s. . .%s %s %s\n"**, <*symbolic−name*>, <*symbolic−name*>, <*encoding*>, <*comments*>

   Example:

   ```
   <U3003>...<U3007>    \x81\x56
   ```

   In this format, the line in the character set mapping definition defines a range of one or more symbolic names. For this form, the **<char_name_mask>** specifies zero or more **A**s, followed by one or more **D**s, or **O**s, or **X**s.

   In this form the line in the character set mapping definition defines a range of symbolic character names and a corresponding encoding. The range is indicated by two symbolic character names separated by ellipsis. The integer formed by the digits in the second symbolic name is equal to or greater than the integer formed by the digits in the first name. This is interpreted as a series of symbolic names formed from the common part and each of the integers between the first and the second integer, inclusive. As an example, **<U3003>...<U3006>** is interpreted as the symbolic names **<U3003>**,**<U3004>**,**<U3005>**, and **<U3006>**, in that order.

3. **"<unassigned> %s. . .%s %s\n"**, <*encoding*>, <*encoding*>, <*comments*>

   Example:

   ```
   <unassigned>          \x9b...\x9c
   ```

   In this form the line in the character set mapping definition defines a single symbolic character name and a range of one or more encoding parts associated with that symbolic character name. This form can only be used with the reserved symbolic character name **<unassigned>**.

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

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

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

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

Decimal constants are represented by two or more decimal digits, preceded by the escape character and the lowercase letter **d**; for example, **\d97** or **\d143.** Hexadecimal constants are represented by two or more hexadecimal digits, preceded by an escape character and the lowercase letter **x**; for example, **\x61** or **\x8f**. Octal constants are represented by two or more octal digits preceded by an escape character.

Each constant represents a single−byte value When constants are concatenated for multibyte character values, they are of the same type, with the last specifies the least significant octet and preceding constants specifies successively more significant octets. Unspecified octets are assigned the value zero; for example, if a 4−octet character is defined as **\x12\x34**, this is equivalent to defining **\x12\x34\x00\x00**.

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

```
<U3003>...<U3007>           \x81\x56
```

is interpreted as:

```
<U3003>                     \x81\x56
<U3004>                     \x81\x57
<U3005>                     \x81\x58
<U3006>                     \x81\x59
<U3007>                     \x81\x5a
```

A range of encoding parts is indicated by an ellipsis separating two encoding parts which are the boundaries of the range For example, the line:

```
<unassigned>                \x9b...\x9c
```

is interpreted as:

```
<unassigned>                \x9b
<unassigned>                \x9c
```

ULS Localization

11 Locales

Locales are defined using the standard **POSIX localedef** syntax. The supplied locales consist of a single **UNIVERSAL** locale with a generic set of attributes and a localized UCS locale for each supported language/territory.

All locale objects are provided in a “UCS” directory under the paths specified by the environment variable **LOCPATH**. For implementations that do not use directories, an implementation defined configuration file provides aliasing and indirection to the actual file names.

11.1 UNIVERSAL Locale

All **ULS** implementations provide a **UNIVERSAL** locale. The behavior of the **ULS** utilities and functions in a **UNIVERSAL** locale is as if the locale was defined via the localedef and layoutdef utiltiies with input from the **UNIVERSAL.src**.

The **UNIVERSAL.src** describe the characteristics and behavior of the **UNIVERSAL** locale for the subset of characters supported from the Basic Multilingual Plane of ISO 10646. On any implementation this subset is equivalent to the union of all charmaps provided. The **UNIVERSAL.src** locale tables can be used in combinations with other national locales to customize the characteristic and behavior of a customize locale. The **UNIVERSAL.src** locale is defined to be independent of language/territory, as much as possible, but to include all the available UCS characters. The **UNIVERSAL** locale only defines the **LC_CTYPE** category. All other categories are obtained from the default values (the same behavior as one would get for the **C** locale).

All implementations define a locale as the default locale, to be invoked when no environment variables are set, or set to the empty string. This default locale can be the **UNIVERSAL** locale or any other, implementation−dependent locale. Some implementations may provide facilities for local installation administors to set the default locale or to customize the set of scripts associated with the **UNIVERSAL** locale. This document does not require such a faciility.

- **UNIVERSAL.src** is the UNIVERSAL locale source file.
- **UNIVERSAL.cm** is the complete default UNIVERSAL charmap file.

  The UNIVERSAL charmap file will be equivalent to specifing all of the charmaps provided by an implementation. As such, the UNIVERSAL charmap can be considered a dynamic subset at any given time.
- **UNIVERSAL.m** is the UNIVERSAL locale methods file.
- **UNIVERSAL** is the name of the UNIVERSAL locale object.

11.2 Localized UCS Locales

For each language/territory, a locale is supplied in which the **UNIVERSAL** locale is customized by applying national locale source and charmap files. These locales are built using the following **localedef** command invocation:

```
localedef −i "Lang_Terr.src UNIVERSAL.src" \
        −f "Local.cm UNIVERSAL.cm" −m UNIVERSAL.m Lang_Terr
```

where:

- **UNIVERSAL.src** , **UNIVERSAL.cm**, and **UNIVERSAL.m** are as defined above for the UNIVERSAL locale.

- ***Lang_Terr*.src** is the language/territory version of the locale source file.
- ***Local*.cm** is a local customized subset of the UNIVERSAL charmap file.
- ***Lang_Terr*** is the localized UCS locale name.

The following table lists a set of possible localized UCS locales and the locale source files and charmaps used to customize the behavior of the **UNIVERSAL** locale.

Locale	Locale Source	Localized Charmap File
ar_AA	ar_AA.src	Arabic.cm
bg_BG	bg_BG.src	Cyrillic.cm
cs_CS	cs_CS.src	Latin−2.cm
da_DK	da_DK.src	Latin−1.cm
da_DK	da_DK.src	Latin−1.cm
de_CH	de_CH.src	Latin−1.cm
de_DE	de_DE.src	Latin−1.cm
el_GR	el_GR.src	Greek.cm
en_GB	en_GB.src	Latin−1.cm
en_KR	en_KR.src	Korean.cm
en_TW	en_TW.src	Taiwan.cm
en_US	en_US.src	Latin−1.cm
es_ES	es_ES.src	Latin−1.cm
fi_FI	fi_FI.src	Latin−1.cm
fr_BE	fr_BE.src	Latin−1.cm
fr_CA	fr_CA.src	Latin−1cm
fr_CH	fr_CH.src	Latin−1.cm
fr_FR	fr_FR.src	Latin−1.cm
hr_HR	hr_HR.src	Latin−2.cm
hu_HU	hu_HU.src	Latin−2.cm
is_IS	is_IS.src	Latin−1.cm
it_IT	it_IT.src	Latin−1.cm
iw_IL	iw_IL.src	Hebrew.cm
ja_JP	ja_JP.src	Japanese.cm
ko_KR	ko_KR.src	Korean.cm
mk_MK	mk_MK.src	Cyrillic.cm
nl_BE	nl_BE.src	Latin−1.cm
nl_NL	nl_NL.src	Latin−1.cm
no_NO	no_NO.src	Latin−1.cm
pl_PL	pl_PL.src	Latin−2.cm
pt_PT	pt_PT.src	Latin−1.cm
ro_RO	ro_RO.src	Latin−2.cm
ru_RU	ru_RU.src	Cyrillic.cm
sh_YU	sh_YU.src	Latin−2.cm
si_SI	si_SI.src	Latin−2.cm
sk_SK	sk_SK.src	Latin−2.cm
sp_YU	sp_YU.src	Cyrillic.cm
sv_SE	sv_SE.src	Latin−1.cm
tr_TR	tr_TR.src	Turkish.cm
zh_TW	zh_TW.src	TChinese.cm

12 Conversion

12.1 Code Set Naming Conventions

TBD

12.2 UCS−2 Conversion Support

All UCS converters are provided in a “uconv” directory under the paths specified by the environment variable **LOCPATH**. For implementations that do not use directories, an implementation−defined configuration file provides aliasing and indirection to the actual file names.

These converters are built using the **uconvdef** command (section 10.3.2). Each converter is capable of supporting conversions in both directions between a particular code set and **UCS−2**. The file name is the name of the code set as described in section 12.1.

12.2.1 Minimum Support

The minimum conversion support required for a ULS implementation on any platform is the support of UCS−2 converters for all code sets supported by that platform. If a platform supports a code set, the implementation of the ULS on that platform must support converters in both direction between that code set and UCS−2.

12.2.2 PC Interoperability Support

The following list of converters

12.2.3 Lists of UCS−2 Interchange Converters

The following tables list the full set of possible UCS−2 two−way converters. ULS implementations may support any subset of these converters, as long as they support the minimum subset described in section 12.2.1.

12.2.3.1 Standard and Open Systems UCS−2 Converters

Converter	Description
ISO8859−1	UCS−2 <−> ISO Latin−1
ISO8859−2	UCS−2 <−> ISO Latin−2
ISO8859−3	UCS−2 <−> ISO Latin−3
ISO8859−4	UCS−2 <−> ISO Latin−4
ISO8859−5	UCS−2 <−> ISO Cyrillic
ISO8859−6	UCS−2 <−> ISO Arabic
ISO8859−7	UCS−2 <−> ISO Greek
ISO8859−8	UCS−2 <−> ISO Hebrew
ISO8859−9	UCS−2 <−> ISO Turkish
JISX0201−0	UCS−2 <−> Japanese JISX0201−0
JISX0208−0	UCS−2 <−> Japanese JISX0208−0
JISX0212−0	UCS−2 <−> Japanese JISX0212−0
CNS11643−1	UCS−2 <−> Chinese CNS11643−1
CNS11643−2	UCS−2 <−> Chinese CNS11643−2
GB2312−0	UCS−2 <−> Chinese GB2312−0
KSC5601−0	UCS−2 <−> Korean KSC5601−0
TIS620−2533−0	UCS−2 <−> Thai TIS620−2533−0
IBM−eucJP	UCS−2 <−> Japanese EUC
IBM−eucKR	UCS−2 <−> Korean EUC
IBM−eucTW	UCS−2 <−> Traditional Chinese EUC
IBM−udcJP	UCS−2 <−> Japanese user−defined characters
IBM−udcTW	UCS−2 <−> Traditional Chinese user−defined characters
IBM−sbdTW	UCS−2 <−> Traditional Chinese IBM−specific characters
UTF−2	UCS−2 <−> UTF−2

12.2.3.2 PC UCS−2 Converters

Converter	Description
IBM−437	UCS−2 <−> USA PC data code
IBM−437−DISPLAY	UCS−2 <−> USA PC display code
IBM−850	UCS−2 <−> Latin−1 PC data code
IBM−850−DISPLAY	UCS−2 <−> Latin−1 PC display code
IBM−852	UCS−2 <−> Latin−2 PC data code
IBM−852−DISPLAY	UCS−2 <−> Latin−2 PC display code
IBM−855	UCS−2 <−> Cyrillic PC data code
IBM−855−DISPLAY	UCS−2 <−> Cyrillic PC display code
IBM−856	UCS−2 <−> Hebrew PC data code
IBM−856−DISPLAY	UCS−2 <−> Hebrew PC display code
IBM−857	UCS−2 <−> Turkish PC data code
IBM−857−DISPLAY	UCS−2 <−> Turkish PC display code
IBM−860	UCS−2 <−> Portuguese PC data code
IBM−860−DISPLAY	UCS−2 <−> Portuguese PC display code
IBM−861	UCS−2 <−> Icelandic PC data code
IBM−861−DISPLAY	UCS−2 <−> Icelandic PC display code
IBM−862	UCS−2 <−> Hebrew PC data code
IBM−862−DISPLAY	UCS−2 <−> Hebrew PC display code
IBM−863	UCS−2 <−> French Canadian PC data code
IBM−863−DISPLAY	UCS−2 <−> French Canadian PC display code
IBM−864	UCS−2 <−> Arabic PC data code
IBM−864−DISPLAY	UCS−2 <−> Arabic PC display code
IBM−865	UCS−2 <−> Nordic PC data code
IBM−865−DISPLAY	UCS−2 <−> Nordic PC display code
IBM−869	UCS−2 <−> Greek PC data code
IBM−869−DISPLAY	UCS−2 <−> Greek PC display code
IBM−910	UCS−2 <−> APL PC code
IBM−932	UCS−2 <−> Japanese PC data code
IBM−932−DISPLAY	UCS−2 <−> Japanese PC display code
IBM−934	UCS−2 <−> Korea PC data code
IBM−934−DISPLAY	UCS−2 <−> Korea PC display code
IBM−936	UCS−2 <−> People’s Republic of China PC data code
IBM−936−DISPLAY	UCS−2 <−> People’s Republic of China PC display code
IBM−938	UCS−2 <−> Taiwan PC data code
IBM−938−DISPLAY	UCS−2 <−> Taiwan PC display code
IBM−942	UCS−2 <−> Extended Japanese PC data code
IBM−942−DISPLAY	UCS−2 <−> Extended Japanese PC display code
IBM−944	UCS−2 <−> Korean PC data code
IBM−944−DISPLAY	UCS−2 <−> Korean PC display code
IBM−946	UCS−2 <−> People’s Republic of China SAA data code
IBM−946−DISPLAY	UCS−2 <−> People’s Republic of China SAA display code
IBM−948	UCS−2 <−> Traditional Chinese PC data code
IBM−948−DISPLAY	UCS−2 <−> Traditional Chinese PC display code
IBM−949	UCS−2 <−> Korean PC data code
IBM−949−DISPLAY	UCS−2 <−> Korean PC display code
IBM−1046	UCS−2 <−> Arabic PC data code
IBM−1046−DISPLAY	UCS−2 <−> Arabic PC display code

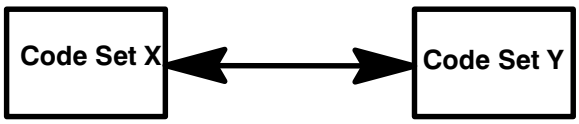
12.2.3.3 EBCDIC UCS−2 Converters

Converter	Description
IBM−037	UCS−2 <−> USA, Canada EBCDIC
IBM−273	UCS−2 <−> Germany, Austria EBCDIC
IBM−277	UCS−2 <−> Denmark, Norway EBCDIC
IBM−278	UCS−2 <−> Finland, Sweden EBCDIC
IBM−280	UCS−2 <−> Italy EBCDIC
IBM−281	UCS−2 <−> Japanese Latin EBCDIC
IBM−284	UCS−2 <−> Spain, Latin America EBCDIC
IBM−285	UCS−2 <−> United Kingdom EBCDIC
IBM−293	UCS−2 <−> APL EBCDIC
IBM−297	UCS−2 <−> France EBCDIC
IBM−420	UCS−2 <−> Arabic EBCDIC
IBM−424	UCS−2 <−> Hebrew EBCDIC
IBM−500	UCS−2 <−> International EBCDIC
IBM−803	UCS−2 <−> Hebrew EBCDIC
IBM−833	UCS−2 <−> Korean EBCDIC
IBM−836	UCS−2 <−> Simplified Chinese EBCDIC
IBM−838	UCS−2 <−> Thai SBCS EBCDIC
IBM−839	UCS−2 <−> Thai EBCDIC
IBM−870	UCS−2 <−> Latin−2 EBCDIC
IBM−871	UCS−2 <−> Icelandic EBCDIC
IBM−875	UCS−2 <−> Greek EBCDIC
IBM−930	UCS−2 <−> Japanese Katakana−Kanji EBCDIC
IBM−933	UCS−2 <−> Korean EBCDIC
IBM−935	UCS−2 <−> Simplified Chinese EBCDIC
IBM−937	UCS−2 <−> Traditional Chinese EBCDIC
IBM−939	UCS−2 <−> Japanese Latin−Kanji EBCDIC
IBM−1025	UCS−2 <−> Cyrillic EBCDIC
IBM−1026	UCS−2 <−> Turkish EBCDIC
IBM−1027	UCS−2 <−> Japanase Latin Extended EBCDIC
IBM−1047	UCS−2 <−> USA, Canada EBCDIC

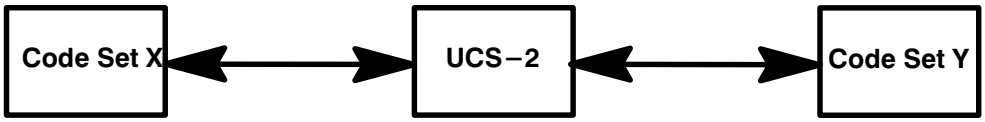
12.3 Additional Conversion Support

In addition to these **UCS−2** interchange converters, the **ULS** may support the converters listed in this section. This can be done by implementing the multibyte **iconv** conversion functions described in section 9.3 so that they automatically look for a pair of UCS−2 converters to perform any conversion operation for which a direct converter is not available.

For example, to provide the following conversion:



the **iconv** functions could actually perform:



This is an economical way to provide many−to−many support for multiple code sets. For example, providing complete interoperability between10 code sets would require 45 different direct converters, but would require only 10 **UCS−2** converters. It is easy to see that the conversion capabilities provided by the above list are

truly formidable. Although many of the conversions would result in significant substitution, one obtains whatever conversion is possible in every case.

12.3.1 Lists of UTF−2 Interchange Converters

The following tables list the set of possible UTF−2 converters. For each of the listed code sets, 2−way conversion capability can be provided between the code set and UTF−2.

12.3.1.1 Standard and Open Systems UTF−2 Converters

Converter	Description
ISO8859−1	UTF−2 <−> ISO Latin−1
ISO8859−2	UTF−2 <−> ISO Latin−2
ISO8859−3	UTF−2 <−> ISO Latin−3
ISO8859−4	UTF−2 <−> ISO Latin−4
ISO8859−5	UTF−2 <−> ISO Cyrillic
ISO8859−6	UTF−2 <−> ISO Arabic
ISO8859−7	UTF−2 <−> ISO Greek
ISO8859−8	UTF−2 <−> ISO Hebrew
ISO8859−9	UTF−2 <−> ISO Turkish
JISX0201−0	UTF−2 <−> Japanese JISX0201−0
JISX0208−0	UTF−2 <−> Japanese JISX0208−0
JISX0212−0	UTF−2 <−> Japanese JISX0212−0
CNS11643−1	UTF−2 <−> Chinese CNS11643−1
CNS11643−2	UTF−2 <−> Chinese CNS11643−2
GB2312−0	UTF−2 <−> Chinese GB2312−0
KSC5601−0	UTF−2 <−> Korean KSC5601−0
TIS620−2533−0	UTF−2 <−> Thai TIS620−2533−0
IBM−eucJP	UTF−2 <−> Japanese EUC
IBM−eucKR	UTF−2 <−> Korean EUC
IBM−eucTW	UTF−2 <−> Traditional Chinese EUC
IBM−udcJP	UTF−2 <−> Japanese user−defined characters
IBM−udcTW	UTF−2 <−> Traditional Chinese user−defined characters
IBM−sbdTW	UTF−2 <−> Traditional Chinese IBM−specific characters

12.3.1.2 PC UTF−2 Converters

Converter	Description
IBM−437	UTF−2 <−> USA PC data code
IBM−437−DISPLAY	UTF−2 <−> USA PC display code
IBM−850	UTF−2 <−> Latin−1 PC data code
IBM−850−DISPLAY	UTF−2 <−> Latin−1 PC display code
IBM−852	UTF−2 <−> Latin−2 PC data code
IBM−852−DISPLAY	UTF−2 <−> Latin−2 PC display code
IBM−855	UTF−2 <−> Cyrillic PC data code
IBM−855−DISPLAY	UTF−2 <−> Cyrillic PC display code
IBM−856	UTF−2 <−> Hebrew PC data code
IBM−856−DISPLAY	UTF−2 <−> Hebrew PC display code
IBM−857	UTF−2 <−> Turkish PC data code
IBM−857−DISPLAY	UTF−2 <−> Turkish PC display code
IBM−860	UTF−2 <−> Portuguese PC data code
IBM−860−DISPLAY	UTF−2 <−> Portuguese PC display code
IBM−861	UTF−2 <−> Icelandic PC data code
IBM−861−DISPLAY	UTF−2 <−> Icelandic PC display code
IBM−862	UTF−2 <−> Hebrew PC data code
IBM−862−DISPLAY	UTF−2 <−> Hebrew PC display code
IBM−863	UTF−2 <−> French Canadian PC data code
IBM−863−DISPLAY	UTF−2 <−> French Canadian PC display code
IBM−864	UTF−2 <−> Arabic PC data code
IBM−864−DISPLAY	UTF−2 <−> Arabic PC display code
IBM−865	UTF−2 <−> Nordic PC data code
IBM−865−DISPLAY	UTF−2 <−> Nordic PC display code
IBM−869	UTF−2 <−> Greek PC data code
IBM−869−DISPLAY	UTF−2 <−> Greek PC display code
IBM−910	UTF−2 <−> APL PC code
IBM−932	UTF−2 <−> Japanese PC data code
IBM−932−DISPLAY	UTF−2 <−> Japanese PC display code
IBM−934	UTF−2 <−> Korea PC data code
IBM−934−DISPLAY	UTF−2 <−> Korea PC display code
IBM−936	UTF−2 <−> People’s Republic of China PC data code
IBM−936−DISPLAY	UTF−2 <−> People’s Republic of China PC display code
IBM−938	UTF−2 <−> Taiwan PC data code
IBM−938−DISPLAY	UTF−2 <−> Taiwan PC display code
IBM−942	UTF−2 <−> Extended Japanese PC data code
IBM−942−DISPLAY	UTF−2 <−> Extended Japanese PC display code
IBM−944	UTF−2 <−> Korean PC data code
IBM−944−DISPLAY	UTF−2 <−> Korean PC display code
IBM−946	UTF−2 <−> People’s Republic of China SAA data code
IBM−946−DISPLAY	UTF−2 <−> People’s Republic of China SAA display code
IBM−948	UTF−2 <−> Traditional Chinese PC data code
IBM−948−DISPLAY	UTF−2 <−> Traditional Chinese PC display code
IBM−949	UTF−2 <−> Korean PC data code
IBM−949−DISPLAY	UTF−2 <−> Korean PC display code
IBM−1046	UTF−2 <−> Arabic PC data code
IBM−1046−DISPLAY	UTF−2 <−> Arabic PC display code

12.3.1.3 EBCDIC UTF−2 Converters

Code Set	Description
IBM−037	UTF−2 <−> USA, Canada EBCDIC
IBM−273	UTF−2 <−> Germany, Austria EBCDIC
IBM−277	UTF−2 <−> Denmark, Norway EBCDIC
IBM−278	UTF−2 <−> Finland, Sweden EBCDIC
IBM−280	UTF−2 <−> Italy EBCDIC
IBM−281	UTF−2 <−> Japanese Latin EBCDIC
IBM−284	UTF−2 <−> Spain, Latin America EBCDIC
IBM−285	UTF−2 <−> United Kingdom EBCDIC
IBM−293	UTF−2 <−> APL EBCDIC
IBM−297	UTF−2 <−> France EBCDIC
IBM−420	UTF−2 <−> Arabic EBCDIC
IBM−424	UTF−2 <−> Hebrew EBCDIC
IBM−500	UTF−2 <−> International EBCDIC
IBM−803	UTF−2 <−> Hebrew EBCDIC
IBM−833	UTF−2 <−> Korean EBCDIC
IBM−836	UTF−2 <−> Simplified Chinese EBCDIC
IBM−838	UTF−2 <−> Thai SBCS EBCDIC
IBM−839	UTF−2 <−> Thai EBCDIC
IBM−870	UTF−2 <−> Latin−2 EBCDIC
IBM−871	UTF−2 <−> Icelandic EBCDIC
IBM−875	UTF−2 <−> Greek EBCDIC
IBM−930	UTF−2 <−> Japanese Katakana−Kanji EBCDIC
IBM−933	UTF−2 <−> Korean EBCDIC
IBM−935	UTF−2 <−> Simplified Chinese EBCDIC
IBM−937	UTF−2 <−> Traditional Chinese EBCDIC
IBM−939	UTF−2 <−> Japanese Latin−Kanji EBCDIC
IBM−1025	UTF−2 <−> Cyrillic EBCDIC
IBM−1026	UTF−2 <−> Turkish EBCDIC
IBM−1027	UTF−2 <−> Japanase Latin Extended EBCDIC
IBM−1047	UTF−2 <−> USA, Canada EBCDIC

12.3.2 List of Converters for Standard, PC and EBCDIC Regional Code Sets

These converters provide conversions for **Standard, PC**, and **EBCDIC** regional code sets. Conversions can be provided between any pair of code sets; however, to avoid significant substitution and data loss, conversions should be restricted to compatible pairs.

The following table lists code sets that are compatible.

Character Set	**Languages**	**PC**	**Standard**	**EBCDIC**
Latin-1	U.S. English, Portugese, Canadian French	IBM-850	ISO8859-1	IBM-037
Latin-1	Danish, Norwegian	IBM-850	ISO8859-1	IBM-277
Latin-1	Finnish, Swedish	IBM-850	ISO8859-1	IBM-278
Latin-1	Italian	IBM-850	ISO8859-1	IBM-280

Latin-1	Japanese	IBM-850	ISO8859-1	IBM-281
Latin-1	Spanish	IBM-850	ISO8859-1	IBM-284
Latin-1	U.K. English	IBM-850	ISO8859-1	IBM-285
Latin-1	German	IBM-850	ISO8859-1	IBM-273
Latin-1	French	IBM-850	ISO8859-1	IBM-297
Latin-1	Belgian, Swiss German	IBM-850	ISO8859-1	IBM-500
Greek	Greek	IBM-869	ISO8859-7	IBM-875
Turkish	Turkish	IBM-857	ISO8859-9	IBM-1026
Arabic	Arabic	IBM-1046 IBM-864	ISO8859-6	IBM-420
Hebrew	Hebrew	IBM-856 IBM-862	ISO8859-8	IBM-424 IBM-803
Latin-2	Eastern European	IBM-852	ISO8859-2	IBM-870
Cyrillic	Eastern European	IBM-855	ISO8859-5	IBM-1025
Japanese	Japanese	IBM-932	IBM-eucJP	IBM-930 IBM-939
Korean	Korean	IBM-934	IBM-eucKR	IBM-933
Traditional Chinese	Traditional Chinese	IBM-938 big-5	IBM-eucTW	IBM-937
Simplified Chinese	Simplified Chinese	IBM-936 big-5	IBM-eucGB	IBM-935

Appendix A: Unicode / ISO10646 Overview

Universal Coded Character Set (UCS) is the name of the ISO 10646 standard that defines a single code for the representation, interchange, processing, storage, entry and presentation of the written form of languages of the world.

The key objectives of this code are to:

- Improve interoperability between systems in an open environment
- Simplify development of internationalized products
- Provide a base for multilingual applications
- Make more characters available

The ISO 10646 has been approved in a formal ISO ballot and published as International Standard, ISO/IEC 10646−1:1993.

13 UCS Architecture

The following sections describe the key features of the ISO IS 10646 architecture.

13.1 Coded Representation Forms

ISO 10646 defines canonical character codes of length 32 bits. This provides code numbers for over 4 billion characters. When used in canonical form to represent text, the coding is referred to as UCS−4 for Universal Coded Character Set 4−byte[10] form.

The code numbers from 0 through 65,535 decimal (FFFF hexadecimal) can be represented by character codes of length 16 bits. When used in this form to represent text, these codes are referred to as UCS−2, for Universal Character Set 2−byte form. This range is also called the *Basic Multilingual Plane (BMP)* of ISO 10646. The standard is arranged so that the most useful characters (covering all major existing standards worldwide) are assigned in this range.

The character code values of UCS−2 are based on Unicode 1.1. Since ISO 10646 does not encode any characters outside of the BMP, the result is that the character repertoires and encoding assignments of Unicode and ISO 10646 are ***absolutely identical.*** For example, consider the character `A (LATIN CAPITAL LETTER A)`, whose code has the unchanging numerical value 65 decimal. This value may be extended by any quantity of leading zeros to serve in the context of the following fixed−length encoding standards:

Standard	Char	Bits	Hex	Decimal
ASCII	A	7	41	65
ISO 8859−1	A	8	41	65
ISO 10646 UCS−2	A	16	0041	65
ISO 10646 UCS−4	A	32	00000041	65

Figure 11. Character Encoding Example

13.2 Implementation Levels

The ISO 10646 encodes a number of **combining** characters (also known as *non−spacing marks* or *floating diacritics*). These characters are necessary in several scripts including Indic, Thai, Arabic, and Hebrew, and can be used also for a productive generation of characters in Latin, Cyrillic, and Greek scripts. However, their presence creates a possibility of an alternative coding for the same text. Although the coding is unambiguous and data integrity is preserved, a processing of text that contains combining characters is more complex. To provide for conformance of applications that choose not to deal with the combining characters, ISO 10646 defines three implementation levels:

- Level 1, which does not allow combining characters
- Level 2, which allows combining marks from Thai, Indic, Hebrew, and Arabic scripts
- Level 3, which allows all combining marks, including the ones for Latin, Cyrillic, and Greek

13.3 Subsets

The ISO 10646 provides the specifications of many subsets and provides ways for identifying them, so that a conforming application or a device does not have to support a full set of characters in the standard. In addition to the explicitly defined subsets, a conforming application can simply list all the characters that it can process. There is no need to code an identification of a subset in the data stream: it can be done by any implicit agreement including documentation.

13.4 ISO 10646 Repertoire and Allocation

The ISO 10646 encodes about **34,200** characters and allocates **6,400** codes in a Private Use Area (PUA). This leaves **24,800** codes on the *BMP* unassigned (38%), which can be used for a future expansion. The distribution of the allocated characters is:

Characters defined in the standard	
Alphabets	2,300
Hangul Jamo alphabet	240
Latin & Greek precomposed forms	478
Symbols	1,376
CJK auxiliaries	842
Hangul syllables	6,656
Unified ideographs	20,902
CJK ideograph compatibility	302
Presentation forms	650
Compatibility and specials	422
Total	***34,180***
Unassigned space	
Private use area	6,400
Reserved for special use	67
Available	24,889

Figure 12. Allocation of characters in ISO 10646

13.5 UCS Transformation Format (UTF−1)

Certain communications systems are sensitive to certain byte values, interpreting them as control instructions. Because ISO 10646 characters can be encoded with any bit combinations in the first or subsequent bytes, ISO 10646 data cannot be freely transmitted through nontransparent systems with the above limitations. To establish a migration path, 10646 defines (in the informative annex) *UCS Transformation Format (UTF−1)* which recodes ISO 10646 data avoiding the usage of C0 controls (0x00...0x1F), C1 controls (0x80...0x9F). space (0x20), and DEL (0x7F). The UTF encoding is a variable length, ranging from 1 to 5 bytes, and is not suitable for a processing code. The UCS to UTF relationship is described in Figure 13:

UCS CHARACTERS			UTF CHARACTERS			
FROM	**TO**	**COUNT**	**1st byte**	**2nd byte**	**3rd byte**	**4th and 5th bytes**
0000 0000	0000 009F	160	00..9F			
0000 00A0	0000 00FF	96	A0	A0..FF		
0000 0100	0000 4015	16,150	A1..F5	21..7E A0..FF		
0000 4016	0003 8E2D	216,600	F6..FB	21..7E A0..FF	21..7E A0..FF	
0003 8E2E	7FFF FFFF	4*190**4	FC..FF	21..7E A0..FF	21..7E A0..FF	21..7E 21..73 A0..FF A0..FF

Figure 13. UCS to UTF relationship

The ASCII characters are encoded in 1 byte, most of the other characters in 2 bytes, and ideographic and compatibility zone in 3 bytes.

14 UTF−2 − A new UCS Transformation Method Proposal

X/Open has developed an alternative proposal for the UCS transformation, primarily oriented to be used in the Unix file systems (the original name of this transformation is FSS−UTF, but it is expected to be registered by ISO as UTF−2). It is expected that UTF−2 will be a dominant transformation method, where UCS is not practical.

The objectives and the definition follow:

With the assumption that most, if not all of the issues surrounding the handling and storing of UCS in historical operating system file systems are understood, the objective is to define a UCS transformation format which also meets the requirement of being usable on a historical operating system file system in a non−disruptive manner. The intent is that UCS will be the process code for the transformation format, which is usable as a file code.

Below are the guidelines that were used in defining the UCS transformation format 2:

1. Compatibility with historical file systems:
   - Historical file systems disallow the null byte and the ASCII slash character as a part of the file name.
2. Compatibility with existing programs:
   - The existing model for multibyte processing is that ASCII does not occur anywhere in a multibyte encoding. There should be no ASCII code values for any part of a

transformation format representation of a character that was not in the ASCII character set in the UCS representation of the character.

3. Ease of conversion from/to UCS.
4. The first byte should indicate the number of bytes to follow in a multibyte sequence.
5. The transformation format should not be extravagant in terms of number of bytes used for encoding.
6. It should be possible to find the start of a character efficiently starting from an arbitrary location in a byte stream.

14.1 The UTF−2 Definition:

The UTF−2 encodes UCS values in the range <0,0x7fffffff> using multibyte characters of lengths 1, 2, 3, 4, 5, and 6 bytes. Single−byte characters are reserved for the ASCII characters in the range <0,0x7f>. These all have the high order bit set to 0. For all characters which are encoded using more than one byte, the number of bytes is indicated by the number of high−order bits which are set to 1. Subsequent bytes of a multi−byte character are all of the form 10xxxxxx.

Bytes	Bits	Hex Min	Hex Max	Byte Sequence in Binary
1	7	00000000	0000007f	0vvvvvvv
2	11	00000080	000007FF	110vvvvv 10vvvvvv
3	16	00000800	0000FFFF	1110vvvv 10vvvvvv 10vvvvvv
4	21	00010000	001FFFFF	11110vvv 10vvvvvv 10vvvvvv 10vvvvvv
5	26	00200000	03FFFFFF	111110vv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv
6	31	04000000	7FFFFFFF	1111110v 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv 10vvvvvv

The UCS value is just the concatenation of the **v** bits in the multibyte encoding. When there are multiple ways to encode a value, only the shortest encoding is legal.

The following is a brief summary of **Unicode.**

15 Unicode

The Unicode character encoding standard is a 16−bit fixed width, uniform text encoding scheme:

15.1 Scope

Unicode defines codes for characters used in all currently significant written languages. In addition to a rich set of scientific and publishing symbols, Unicode covers the following scripts.

Arabic, Armenian, Bengali, Bopomofo, Cyrillic, Devanagari, Georgian, Greek, Gujarati, Gurmukhi, Hangul, Chinese Hanzi, Hebrew, Hiragana, IPA (International Phonetic Alphabet), Katakana, Japanese Kanji, Kannada, Korean Hanja, Lao, Latin, Malayalam, Oriya, Tamil, Telugu, Thai, and Tibetan.

Unicode defines an implicit algorithm for uniquely interpreting bidirectional (Arabic and Hebrew) character data streams. It also incorporates several formatting characters for explicit manipulation of the bidirectional data.

15.2 The Unicode Consortium

Unicode is developed and maintained by a consortium of software and hardware manufacturers including:

Adobe, Apple, Borland, Digital, GO, HP, IBM, Lotus, Microsoft, NeXT, Novell, Research Libraries Group, Sun, Taligent, Unisys, WordPerfect, and Xerox.

15.3 Unicode Availability

”Unicode 1.0” is available from Addison−Wesley in two volumes.

- Volume 1 − non−ideographic part (ISBN 0−201−56788−1)
- Volume 2 − ideographic part (ISBN 0−201−60845−6)

15.4 Unicode 1.01 and Unicode 1.1

Although 10646 UCS−2 was based on Unicode 1.0, it was not identical. The Unicode Consortium has issued a version 1.1 of Unicode, which fully aligns with UCS−2 making the changes and additions agreed by ISO. Unicode 1.1 was issued as a technical report during 2Q 1993.

In order to expedite use of Unicode in the interim, the Unicode consortium issued an intermediate version, Unicode 1.0.1, which consisted of Unicode 1.0 modified by the changes necessary to make the character codes a proper subset of ISO 10646. Implementations that used Unicode 1.0.1 are completely compatible with Unicode 1.1, and therefore fully compatible with ISO 10646.

16 Unicode and ISO 10646 Scope Differences

In addition to the temporary differences between Unicode 1.0 and 10646 UCS−2, these standards also differ in several other, more permanent, aspects. They are briefly described in the following sections.

16.1 Character Attributes

ISO 10646 scope is primarily to assign a code to each character in the standard. In addition to codes, Unicode defines various character properties such as digits, spaces, and others. It also defines how the characters are used and how they relate to each other. This information provides background material to help Unicode implementers.

16.2 Bidirectional Architecture

Unicode defines a number of formatting controls to control the presentation of text written in Arabic, Hebrew, or any other language where display order of the characters is mixed: national characters are presented from right to left, and digits (or imbedded Latin text) from left to right. Unicode also defines the rules for precise interpretation of plain bidirectional text (called “implicit algorithm”) in the absence of the formatting controls. ISO 10646 includes all of the formatting controls, but does not define the implicit algorithm, leaving the freedom to the implementers. Thus, Unicode implementation of Arabic or Hebrew will conform with ISO 10646, but the reverse is not necessarily true.

16.3 Conformance

The Unicode conformance clause is quite simple: “For each 16−bit character code in a text sequence, a process that conforms to the Unicode standard must either interpret the code value according to its Unicode semantics, or not interpret it at all. An interchange process may not change a code that it cannot interpret.” Unicode does not require an application to support all of the characters, and does not identify any particular subsets. Each application is free to select its own subset and the ways to identify it, if necessary.

An ISO 10646−conforming application is required to specify the coded representation form (UCS−2 or UCS−4), the implementation level (level 1 with precomposed characters only, or levels 2 and 3 that may include combining marks), and the adopted subset.

Any Unicode application will conform to ISO 10646 by specifying UCS−2, Level 3, Level 2, or Level 1, and any appropriate subset of the ISO 10646.

17 What does it mean to implement Unicode

Simply put, it means to use Unicode for all of the textual data or some part of it. Although Unicode contains characters for many languages, it is not necessary to support all of these languages when implementing it. It is anticipated that the advantages of a single character code for interoperability and national language support will lead to a critical mass of applications and devices supporting Unicode in the not−too−distant future. On the other hand, a native support for Unicode is a significant investment in software and hardware which will not happen overnight. However, there are two main implementation scenarios which can immediately benefit from using Unicode. They can be defined as an interchange model and a processing model.

17.1 Unicode as interchange code

In this model, two or more local systems interchange data via communication lines or any other media. Using Unicode as interchange code means that each system will convert a local code to Unicode when sending data, and from Unicode to a local code when receiving data. Some of the benefits and costs associated with this model are described below:

17.1.1 Benefits

- Reduce the complexity

  Naturally each system knows its own encoding scheme(s). If only Unicode is used as the interchange code, a system will not have to deal with a complexity of multiple 'foreign' encodings. For example, EBCDIC hosts will not have to deal with various SJIS, EUC or T.61 encodings. Similarly, PC systems will not have to deal with various EBCDIC encodings.

- Reduce server resource requirements

  To support a variety of encodings the server will need to install and maintain many tables. If 50 different code sets are allowed as network codes in a heterogeneous network, (not an unreasonable assumption, in view of the number of existing code sets). Each server would need 49 tables to support every local encoding. This is compared to just one table, if Unicode is used as the network code. The size of the conversion table from Unicode will reflect the capacity of the local encoding and the density of these characters' allocation in Unicode. So, for example, if a system only supports a single−byte code set, the table does not need to be huge.

- Simplify the maintenance of the network

  Suppose a new client using encoding X joins the network. If Unicode is used as a network code, this client has to convert from X to Unicode only. Servers do not need any changes. If X is allowed to become an additional network code, each server on the network has to add the parsing capability and the conversion table(s) from X to its local codes.

- Improve interoperabiltiy

  It is true that if a terminal supports only English, using Unicode as a network code will not help it process Russian or Japanese. But in a real distributed system with multiple network codes, the likelihood of every server being able to deal with every encoding at any time is remote. If an EBCDIC system does not recognize a Russian PC code set, it cannot process a message which was encoded in this code even if, in fact, the actual text contained only Latin characters. Similarly, two systems may not be able to communicate, because they do not recognize the code sets of each other, even if these code sets define the same characters. All of the above would not happen if Unicode were the network code.

17.1.2 Costs

- Packet size

  Using Unicode as a network code can double the size of communication packets especially for Europe and North America. The impact will be reduced if compression schemes are utilized, but at the additional cost of conversion time during send/receive operations. A study is required to estimate the impact and feasibility of compression in practical situations.

- Conversion time

  Using Unicode as a network code implies conversions at both client and server ends of the communication (unless a local system is using Unicode as its native code). Particularly if, in fact, the communication is between homogeneous systems, these two conversions could be saved if the data packet were tagged instead of employing Unicode. In many simple cases however, the performance impact of the character data conversion is small. A study is required to estimate the performance impact in practical situations.

Using Unicode as an interchange code is particularly attractive in global, multivendor, heterogenous networks. Unicode is gaining a significant level of support in the vendor community. Systems will have to be tolerant of these environments to continue to interoperate with them

17.2 Unicode as a processing code

Using Unicode as a processing (or internal) code is in a sense the opposite model to the interchange model. An application may transform all its internal processing and file codes to Unicode and convert to or from other codes when communicating with other applications or devices (with the outside world) which encode character data in codes other than Unicode. This model is possible if there are clearly indentifiable conversion points between the domain where other codes are used.

Within the Unicode domain, the application has to deal only with one single and uniform code for all the supported languages. The major benefit is being able to add support for additional languages without changing the main body of the code. This applies both to an application that supports each language separately (unilingual) or in parallel (multilingual).

A local system that uses Unicode will gain an additional benefit if it participates in the network where Unicode is used as interchange code.

Appendix B: Writing System Analysis

This section contains an analysis of the world's writing systems and identifies the main features of written languages that a computing system should support. We consider only writing systems that have a publishing tradition and hence exclude systems such as ancient Greek written boustrophedan style, in which the writing direction is changed at the end of each line, medieval Islamic Nast'liq in which each word is written at an angle of approximately 45 degrees, and Cheyenne picture writing.

The table in figure 14 lists the scripts, the languages written in the scripts and the approximate number of speakers. The figures for the number of speakers should be treated with caution. For example, they include people of all ages and take no account of literacy rates or the ability to speak two or more languages. A second language is particularly important in India where English is commonly used for business correspondence. Extrapolating from this data to a population market would require additional demographic, educational and economic data.

It is important to note that Figure 14 illustrates that there is not a one−to−one correspondence between script and geographic area or even between script and language family: for instance, Urdu, an Indoeuropean language that is geographically based in Pakistan, uses an Arabic−based script. Farsi, also an Indoeuropean language geographically based in Iran, also uses an Arabic script. In addition, the spread of cultures throughout the world results in multilingualism and the use of languages and scripts in geographic areas far removed from their geographic area of origin.

Type of Script	Script and Languages (speakers in thousands) *	Total * (Millions)
1 Roman type: vowels inde− pendent signs.	***Roman Script, total=1,725 M, used for:*** English(350000), Spanish(275000), Indonesian(165000), Portuguese(125000), German(100000), French(75000) Italian(60000), Vietnamese(55000), Javanese(50000), Turkish(50000), Slovak(45000), Polish(35000), Flemish(20500), Hausa(20000), Rumanian(20000), Ibo(15000), Sundanese(15000), Yoruba(15000), Chuang(12000), Malagasy(10000), Swahili(10000), Czech(10000), Tagalog(10000), Hungarian(10000), Swedish(8300), Visayan(8000), Madurese (8000), Somali(7000), Malay(7000), Faluni(6000), Catalan(6000), Twi(6000), Croation(6000), Shona(5500), Yi (Lolo) (5000), Danish(5000), Finnish(5000), Ruanda(5000), Albanian(4000), Xhosa(4000), Zulu(4000), Afrikaans(4000), Rundi(4000), Norwegian(4000), U Luba(3000), Mossi(3000), Sotho(3000), Tswana(3000), Ganda(3000), Lithuanian(2500), Wolof(2500), Nyanja(2500), Malinke(2000), Kongo(2000), Latvian(1500), Slovenian(1500), Ewe(1500), Fang(1500), Bemba(1500), Swazi(1100), Mende(1000), Estonian(1000), Basque(700), Welsh (500), Icelandic(250) ***Cyrillic script, total=265 M, used for:*** Russian(153000), Ukranian(35000). Uzbek(12000), Serbian(11000), Bulgarian(8000), Belorussian(7000), Tater(6000), Azerbaijani(5000), Kazakh(5000), Mongolian(4500), Tadzhik(3000), Moldavian(2500), Kirgiz(2000), Turkmen(2000), Chuvash(1500), Bashkir(1000), Macedonian(1000), Udmurt(1000), Mordvin(1000), Chechen(750), Mari(500), Ossetian(500), Avar(500), Kabardian(360), Yakut(300), Buryat(300), Komi(275), Kalmyk(125), Nenets(25), Abaza(25), Evenki(15) ***Following languages, total=34 M, have their own scripts:*** Amharic(10000), Greek(10000), Armenian(4000), Miao(4000), Georgian(3500), Buginese(2500)	2,024
2 Logographic	Following languages have their own (similar) scripts: Chinese(1030000), Japanese(120000), Korean(60000)	1,210
3 Indic • Southern (Dravidian) • Northern (Nagari based)	SOUTHERN ***Following languages, total=293 M, have their own scripts:*** Telugu(60000), Tamil(54000), Thai(45000), Burmese(30000), Kanarese(30000), Malayalam(30000), Sinhalese(11000), Oriya(25000), Khmer(5000), Lao(3000) NORTHERN ***Devenagari script, total=335 M, used for:*** Hindi(225000), Marathi(55000), Bihari(40000), Nepali(12000), Kashmiri(3000), (Urdu is an Arabic−based script in Pakistan), Sanskrit(). ***Following languages, total=283 M, have their own scripts:*** Bengali(160000), Punjabi(75000), (script is***Gurmukhi***), Gujarati(35000), Assamese(12000), Tibetan(1500)	911
4 BiDirectional	***Arabic Script, total=282 M, used for:*** Arabic(150000), Farsi(35000), Urdu(35000), Pashto(10000), Sindhi(7000), Kurdish(5000) (Cyrillic in USSR, Roman in Iraq and Syria), Uigur(5000) (Cyrillic in USSR) Kasakh(now Cyrillic), ***Hebrew Script***: Hebrew(3000)	285
Total: 4,431 Million People (covering 136 languages and 31 scripts) * Source: *The Languages of the World*, Kenneth Katzner (1986)		

Figure 14. Languages of the world, their speakers and type of script.

By grouping the scripts according to their formatting characteristics we are able to show that four basic script types account for the world's scripts. There is no doubt that the inclusion of Mongolian, and alphabetic scripts written vertically, would require a fifth script type. Mongolian was excluded because it is of little commercial value at this point (Cyrillic is also used to write the language) and because our architecture should allow the addition of further script classes. The point is not that there are four or five script types but that very few types account for a great many languages.

The following lists the scripts a multilingual system should be able to support.

Type of Script	Script name	Type of Script	Script name
LOGOGRAPHIC	Chinese Japanese Korean	RIGHTtoLEFT	Arabic Hebrew
INDIC(Southern)	Bengali Burmese Kanarese Khmer Lao Malyalam Oriya Sinhalese Tamil Telugu Thai	INDIC(Nothern)	Devanagari Gujariti Gurmukhi Tibetan
LATIN	Amharic Armenian Buginese Cyrillic Georgian Greek Latin varients: West European East European Oriental African Miao	OTHER (Latin)	Chemistry Math Symbols(publishing) Phonetic User (not mapped)

Figure 15. World Scripts

The following shows some of the characteristics of each script type as they affect fonts, input and output.

Type of Script	Input Method Feature†	Output Method Features‡	Font Features
1 *Latin type Vowels and consonants separate (e.g. Latin, Cyrillic, Greek)*	• Must handle diacritics so that e+′ generates code point for é. • Select keyboard mode (Cyrillic, Greek, Russian...)		Ligatures required in publications.
2. *Logographic* (e.g. scripts for Chinese, Korean and Japanese)	• Phonetic conversion e.g. Romanji, Katakana,or Hiragana to Kanji. • Select keyboard mode (Romaji, Hiragana, Katakana,etc.)	Large character set may require special processing.	• Kerning to generate Kanji built from ”radicals”. • Vertical writing support
3 *Indic* Southern: Vowel above, below and beside consonant (e.g. Tamil, Oriya) Northern: Vowels, modify consonants (e.g. Devanagari, Bengali)	Choice of keyboard, either phonetic (ISCII) or graphical (typewriter)	• Reverse certain phonetic CV ordering to VC. • Perform character shaping, depending on type of datastream.	• Kerning for vowel marks (matras) • Ligatures in publications • Ligatures for ”Conjoints”
4. *BiDirectional* (e.g.Arabic, Hebrew) Vowels above, below and beside consonants	• Set input mode (push mode) • Generate required datastream (explicit, implicit, visual)	• RtL and LtR processing • Character shaping; present initial, medial, final or isolated form according to context.	• Kerning useful for presenting vowels (graphics only?). • Ligatures required (400+,3 mandatory)
† Keyboard layering required for all of these.			
‡ Each output method requires special process to compute word breaks and the length of formatted strings.			

Figure 16. Processing requirements for each script type

Appendix C: ULS Programming Tips

18 XPG/ISO−C vs ULS Function Comparison

		XPG/ISO−C		ULS	
	SBCS	Multibyte	wchar_t	UniChar	
Locale Management		setlocale nl_langinfo localeconv		✓	
Convert mb<−>wc		mbtowc mbstowcs	wctomb wcstombs	(see Convert Codeset)	
Classification	isalpha is*		iswalpha isw*	✓	
Case Mapping	tolower toupper		towlower towupper	✓	
Time/Monetary		strftime, strptime strfmon	wcsftime	✓	
String Copy		strcat strcpy	wcscat wcsncat wcscpy wcsncpy	✓	
String Collate		strcoll strxfrm	wcscoll wcsxfrm	✓	
String Length		strlen	wcslen	✓	
String Search	strchr strcspn strpbrk strrchr strspn strtok		wcschr wcscspn wcspbrk wcsrchr wcsspn wcstok	✓	
IO Display Width			wcwidth wcswidth	✓	
IO Printf		printf, vprintf sprintf, vsprintf, fprintf, vfprintf	printf, vprintf sprintf, vsprintf fprintf, vfprintf	✓	
IO Scanf		scanf sscanf fscanf	scanf sscanf fscanf	✓	
IO Chararacter	getc, gets putc, puts		fgetwc, fgetws fputwc, fputws	✓	
Message		catopen catgets catclose		??	
Convert Codeset		iconv_open iconv iconv_close		✓	

Figure 17. Summary of ULS Functions

19 UCS Processing

20 UCS Intechange

21 UCS Presentation / Input

22 UCS File Storage

Appendix D: ULS Input and Output

23 Text Data Object and I/O Functions

23.1 Text Data Object Functions

The text data object access functions provide the interfaces to manipulate text data objects. Text data objects can be used for general text data manipulation including input and/or output operations. All input and output operations are performed using a text data object.

The **UniCreateTextDataObject()** function allocates resources associated with the specified text data object and provides the caller with a text data object handle. The text data object contains all of the information needed to perform I/O operations including code set conversion.

The **UniQueryTextDataObjectLocale()** function obtains information about the locale object of a particular text data object.

The **UniQueryTextDataObjectStream()** function obtains information about the file stream of a particular text data object.

The **UniQueryTextDataObjectConv()** function obtains information about the conversion associated with a particular text data object.

The **UniFreeTextDataObject()** function deallocates resources associated with text data objects.

23.1.1 UniCreateTextDataObject() – Create a text data object

```
#include <UniChar.h>
int UniCreateTextDataObject(
        const LocaleObject locale_object,
        int type,
        const void *stream_info,
        const UconvObject uconv_object,
        TextDataObject *text_data_object
        );
```

The **UniCreateTextDataObject()** function creates a text data object according to the passed arguments. The object created is an opaque object containing the information necessary to perform input and output operations and other text manipulations. The pointer to the text data object is returned in **text_data_object**.

The **locale_object** argument, which is a locale object handle returned by the **UniCreateLocaleObject()** function, specifies the locale object to be associated with the text data object. The **locale_object** argument should not be a NULL pointer.

The **type** argument supplies information about the **stream_info** argument and text data conversion. The **type** argument is constructed by or–ing flags from the following list:

UNI_FILE	The **stream_info** argument points to a **FILE** stream type. This value and the UNI_HFILE value are mutually exclusive.
UNI_HFILE	The **stream_info** argument points to an **HFILE** stream type. This value and the UNI_FILE value are mutually exclusive.
UNI_DEFAULT	Use the platform default code set for text data conversion. The value of the **uconv_object** is ignored. This value and the UNI_PASS_THRU value are mutually exclusive.
UNI_PASS_THRU	Do not perform a conversion on text data. The value of the **uconv_object** is ignored. This value and the UNI_DEFAULT value are mutually exclusive.

The **stream_info** argument points to the stream type.

The **uconv_object** argument, which is an object handle returned by the **UniCreateUconvObject()** function, specifies the type of conversion to be performed.

The **UniCreateTextDataObject()** function returns one of the following values:

UNI_SUCCESS	the requested text data object was created.
UNI_NOMEMORY	there is insufficient memory to create the requested text data object; the text data object pointer points to undefined data.
UNI_INVALID	an invalid **type** argument specification; the text data object pointer points to undefined data.

23.1.2 UniQueryTextDataObjectLocale() – Query text data object locale data

```
#include <UniChar.h>
int UniQueryTextDataObjectLocale(
        TextDataObject text_data_object,
        LocaleObject *locale_object
        );
```

The **UniQueryTextDataObjectLocale()** function returns a pointer to a locale specification in the area pointed to by **locale_object**. The locale object pointer retrieved is the same pointer that was passed to the **UniCreateTextDataObject()** function when the text data object was created.

If **text_data_object** is invalid, the contents of **locale_object** are undefined.

The **UniQueryTextDataObjectLocale()** function returns one of the following values:

UNI_SUCCESS	locale object pointer successfully retrieved.
UNI_BADOBJ	invalid text data object specified.

23.1.3 UniQueryTextDataObjectStream() – Query text data object stream data

```
#include <UniChar.h>
int UniQueryTextDataObjectStream(
        TextDataObject text_data_object,
        int *stream_type,
        void *stream_info
        );
```

The **UniQueryTextDataObjectStream()** function returns a pointer to a stream type in the area pointed to by **stream_info** qualified by the value returned in **stream_type**. The stream type pointer and the stream type value retrieved are the same values that were passed to the **UniCreateTextDataObject()** function when the text data object was created.

If **text_data_object** is invalid, the contents of **stream_type** and **stream_info** are undefined.

The **UniQueryTextDataObjectStream()** function returns one of the following values:

UNI_SUCCESS	file stream information successfully retrieved.
UNI_BADOBJ	invalid text data object specified.

23.1.4 UniQueryTextDataObjectConv() – Query text data object conversion data

```
#include <UniChar.h>
int UniQueryTextDataObjectConv(
        TextDataObject text_data_object,
        UconvObject *uconv_object
        );
```

The **UniQueryTextDataObjectConv()** function returns a pointer to a conversion object in the area pointed to by **uconv_object**. The conversion object pointer retrieved are the same values that were passed to the **UniCreateTextDataObject()** function when the text data object was created.

If **text_data_object** is invalid, the contents of **uconv_object** are undefined.

The **UniQueryTextDataObjectConv()** function returns one of the following values:

UNI_SUCCESS	conversion object pointer successfully retrieved.
UNI_BADOBJ	invalid text data object specified.

23.1.5 UniFreeTextDataObject() – Free text data object

```
#include <UniChar.h>
int UniFreeTextDataObject(
        TextDataObject text_data_object
        );
```

The **UniFreeTextDataObject()** function destroys the text data object identified by **text_data_object** and frees any memory associated with it.

The **UniFreeTextDataObject()** function returns one of the following values:

UNI_SUCCESS	the specified text data object successfully destroyed.
UNI_BADOBJ	the text data object specified by **test_data_object** is not a valid text data object.

23.2 Input/Output Functions

23.2.1 UniInput() − Input character string

The input function reads from the stream and returns Unicode character codes. The behavior of input is affected by the information contained in the text data object as indicated by the text data handle argument. Read errors refer to both conversion failures and failures of the underlying read operations.

```
#include <UniChar.h>
int UniInput(
        const TextDataObject text_data_object,
        UniChar *ucs,
        int n,
        int *num_read
        );
```

The **UniInput()** function reads code elements (at most one less than the number of the code elements specified by **n**) from the stream associated with **text_data_object** into the array pointed to by **ucs**. No additional code elements are read after a new−line character (which is retained) or after end−of−file. A null code element is written immediately after the last code element read into the array. If an error occurs during the operation, the array contents are indeterminate.

On completion of a successful input operation, **num_read** contains a count of the number of code elements stored at **ucs** excluding the terminating null code element.

The **UniInput()** function returns one of the following:

UNI_SUCCESS	Input operation completed successfully.
UNI_BADF	The file descriptor underlying stream is not a valid file descriptor open for reading.
UNI_INTR	The read operation was terminated due to an interruption.
UNI_NOMEMORY	Insufficient storage space is available.
UNI_NOXIO	A request was made of a non−existent device, or the request was outside the capabilities of the device.
UNI_ILSEQ	The data obtained from the input stream do not form a valid code element.

23.2.2 UniOutput() – Output a character string

The output function performs any necessary conversion of the Unicode character codes and writes the result to the stream. The behavior of the output function is affected by the information contained in the text data object as indicated by the text data object handle argument. Write errors refer to both conversion failures and failures of underlying write operations.

```
#include <UniChar.h>
int UniOutput(
        const TextDataObject text_data_object,
        const UniChar *ucs,
        int n,
        int *num_written
);
```

The **UniOutput()** function writes up to **n** code elements from the string pointed to by **ucs** to the stream associated with the text data object pointed to by **text_data_object**. When the value of **n** is 0, code elements are written until a null code element is encountered. When a null code element is encountered before **n** code elements have been written, the output to the stream is terminated. The terminating null character is not written. Upon successful completion the **UniOutput()** function **num_written** contains the number of code elements written.

The **UniOutput()** function returns one of the following:

UNI_SUCCESS	Output operation completed successfully.
UNI_BADF	The file descriptor of the underlying stream is not a valid file descriptor open for writing.
UNI_FBIG	An attempt was made to write to a file that exceeds the process's file size limit or the maximum file size.
UNI_INTR	The write operation was terminated due to an interruption.
UNI_NOSPC	There was no free space remaining on the device containing the file.
UNI_PIPE	An attempt is made to write to a pipe that is not open for reading by any process.
UNI_NOMEMORY	Insufficient storage space is available.
UNI_NOXIO	A request was made of a non−existent device, or the request was outside the capabilities of the device.
UNI_ILSEQ	A code element does not correspond to a valid character.

23.2.3 UniUnGet() – Push a character onto a stream

```
#include <UniChar.h>
int UniUnGet(
        const TextDataObject text_data_object,
        int uc
        );
```

The **UniUnGet()** function pushes the code element specified by **uc** back onto the input stream associated with the text data object pointed to by **text_data_object**. The pushed–back code elements will be returned by subsequent reads on that stream in the reverse order of their pushing. A successful intervening call to a file positioning function discards any pushed–back characters for the stream. The external storage corresponding to the stream is unchanged. One code element of push–back is guaranteed provided something has been read from the stream and the stream is buffered. If the **UniUnGet()** function is called too many times on the same stream without an intervening read or file positioning operation on that stream, the operation may fail. If the value of **uc** equals 0xFFFF, the operation fails and the input stream is unchanged. A successful call to the **UniUnGet()** function clears the end–of–file indicator for the stream. The value of the file position indicator for the stream after reading or discarding all pushed–back code elements shall be the same as it was before the code elements were pushed back. For a text stream, the value of its file position indicator after a successful call to the **UniUnGet()** function is unspecified until all pushed–back code elements are read or discarded. For a non–text stream, the value of its file position indicator is undefined; if its value was zero before a call, it is indeterminate after the call

The **UniUnGet()** function returns one of the following:

UNI_SUCCESS	Unget operation completed successfully.
UNI_EOF	Attempted to write 0xFFFF to the stream.
UNI_ILSEQ	The code element does not correspond to a valid character.
UNI_NOSPC	There was no space to allow push–back of the the code element to the stream.

23.3 Formatted Input/Output Functions

The formatted input/output functions include a text data object handle argument which determines the behavior for the duration of the function call. The format strings and conversion specifiers of both the **UniFprintf** and **UniFscanf** families of functions accept only a **UniChar** string data type.

23.3.1 UniFprintf() – Write a formatted string to a stream

```
#include <UniChar.h>
int UniFprintf(
        const TextDataObject text_data_object,
        const UniChar *format,
        ...
        );
```

The **UniFprintf()** function writes output to the stream associated with the text data object pointed to by **text_data_object**, under control of the code element string pointed to by **format** that specifies how subsequent arguments are converted for output. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored. The function returns when the end of the format is encountered.

Conversions can be applied to the nth argument after the **format** argument list, rather than to the next unused argument. In this case, the conversion character **%** (see below) is replaced by the sequence **%n$**, where **n** is a decimal integer in the range [1,(NL_ARGMAX)], giving the position of the argument in the argument list. This feature provides for the definition of format strings that select arguments in an order appropriate to specific natural languages.

In format strings containing the **%n$** form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings which use the **%** form of conversion specifications, each argument in the argument list is used exactly once.

It is not valid to intermix format strings with **%** forms of conversion specifications and **%n$** forms of conversion specifications, except that **%%** can be intermixed with **%n$** conversion specifications..

Note that all characters in the format that are described as special characters in the following description (such as space and **%**) are members of the basic (portable) character set.

The format is composed of zero or more directives: ordinary characters (not **%**), which are written to the output stream; and conversion specifications, each of which results in fetching zero or more subsequent arguments. Each conversion specification is introduced by the character **%** or by the character sequence **%n$**., after which 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 code elements than the field width, it will be 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 code elements 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 behavior is undefined.

- 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 will have been promoted according to the integral promotions, and its value shall be 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 (ell) 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 behavior is undefined.

- A 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. In format strings containing the **%n$** form of a conversion specification, a field width or precision may be indicated by the sequence ***m$**, where **m** is a decimal integer in the range [1,(NL_ARGMAX)] giving the position of the argument list (after the **format** argument) of an integer argument containing the field width or precision.

The flag characters and their meanings are

− The result of the conversion will be left−justified within the field. (It will be right−justified if this flag is not specified.)

+ The result of a signed conversion will always begin with a plus or minus sign. (It will begin with a sign only when a negative value is converted if this flag is not specified.)

space If the first character of a signed conversion is not a sign, or if a signed conversion results in no characters, a space will be prefixed to the result. If the **space** and + flags both appear, the **space** flag will be 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 nonzero result will have 0x (or 0X) prefixed to it. For e, E, f, g, and G conversions, the result will always contain a radix character, even if no digits follow it. (Normally, a radix character appears in the result of these conversions only if a digit follows it.) For g and G conversions, trailing zeros will not be removed from the result. For other conversions, the behavior 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 will be ignored. For d, i, o, u, x, and X conversions, if a precision is specified, the 0 flag will be ignored. For other conversions, the behavior 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 will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no 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 will be expanded with leading zeros. The default precision is 1. The result of converting a zero value with a precision of zero is no characters.

f The double argument is converted to decimal notation in the style [−]ddd.ddd, where the number of digits after the radix 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 radix character appears. If a radix 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±dd, where there is one digit before the radix character (which is nonzero if the argument is nonzero) 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 radix character appears. The value is rounded to the appropriate number of digits. The E conversion specifier will produce 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) will be 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 radix character appears only if it is followed by a digit.

c The argument is converted to a character and written.

s The argument shall be a pointer to an array of UniChar. Code elements from the array are written up to (but not including) a terminating null code element; if the precision is specified, no more than that many code elements are written. If the precision is not specified or is greater than the size of the array, the array shall contain a null code element.

P The argument shall be a pointer to void. The value of the pointer is converted to a sequence of printable characters.

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

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

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

In no case does a nonexistent 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.

The **UniFprintf()** function returns the number of code elements transmitted, or a negative value if an output error occurred.

23.3.2 UniSprintf() – Write a formatted string to memory

```
#include <UniChar.h>
int UniSprintf(
        const TextDataObject text_data_object,
        UniChar *ucs,
        size_t n,
        const UniChar *format,
        ...
        );
```

The **UniSprintf()** function is equivalent to the **UniFprintf()** function except that the **ucs** argument specifies an array of **UniChar** into which the generated output is to be written, rather than to a stream. No more than **n** code elements are written including a terminating null code element. A null code element is written at the end of the code elements written; it is not counted as part of the returned sum. If copying takes place between objects that overlap, the behavior is undefined.

The **UniSprintf()** function returns the number of code elements written in the array, not counting the terminating null code element.

23.3.3 UniFscanf() – Read a formatted string from a stream

```
#include <UniChar.h>
int UniFscanf(
        const TextDataObject text_data_object,
        const UniChar *format,
        ...
        );
```

The **UniFscanf()** function reads input from the stream associated with the text data object pointed to by **text_data_object** under control of the **UniChar** string pointed to by **format** that specifies the admissible input sequences and how they are to be converted for assignment, using subsequent arguments as pointers to the objects to receive the converted input. If there are insufficient arguments for the format, the behavior is undefined. If the format is exhausted while arguments remain, the excess arguments are evaluated but are otherwise ignored.

Conversions can be applied to the nth argument after the **format** argument list, rather than to the next unused argument. In this case, the conversion character **%** (see below) is replaced by the sequence **%n$**, where **n** is a decimal integer in the range [1,(NL_ARGMAX)]. This feature provides for the definition of format strings that select arguments in an order appropriate to specific natural languages.

In format strings containing the **%n$** form of conversion specifications, numbered arguments in the argument list can be referenced from the format string as many times as required.

In format strings which use the **%** form of conversion specifications, each argument in the argument list is used exactly once.

It is not valid to intermix format strings with **%** forms of conversion specifications and **%n$** forms of conversion specifications, except that **%%** or **%*** can be intermixed with **%n$** conversion specifications..

Note that all code elements in the format that are described as special characters in the following description (such as white−space and **%**) are members of the basic (portable) character set.

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

- An optional assignment−suppressing character *.
- An optional nonzero decimal integer that specifies the maximum field width.
- An optional l (ell) or L indicating the size of the receiving object. 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 (ell) 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 (ell) if it is a pointer to unsigned long int. Finally, the conversion specifiers e, f and g shall be preceded by l (ell) 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 doubles. If an h, l (ell) or L appears with any other conversion specifier, the behavior is undefined.
- A conversion character that specifies the type of conversion to be applied. The valid conversion specifiers are described below.

The **UniFscanf()** function executes each directive of the format in turn. If a directive fails, as detailed below, the 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 character(s) is executed by reading input up to the first non−white−space character (which remains unread), or until no more characters can be read.

A directive that is an ordinary character is executed by reading the next character of the stream. If the character differs from the directive, the directive fails, and the differing and subsequent 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 characters (as specified by the space attribute) are skipped, unless the conversion specification includes a [, c, C or n specifier. These white−space characters are not counted against a specified field width.

An input item is read from the stream, unless the specification includes an specifier. An input item is defined as the longest matching sequence of input characters, unless that exceeds a specified field width, in which case it is the initial subsequence of that length in the sequence. The first 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 an 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 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 behavior 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 **UniStrtol()** function with the value 10 for the 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 **UniStrtol()** function with the value 0 for the base argument. The corresponding argument shall be a pointer to integer.

o — Matches an optionally signed octal integer, whose format is the same as expected for the subject sequence of the **UniStrtoul()** 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 **UniStrtoul()** 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 **UniStrtoul()** 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 UniChar string of the **UniStrtod()** function. The corresponding argument shall be a pointer to floating.

s — Matches a sequence of non−white−space characters. The corresponding argument shall be a pointer to the initial code element of an array large enough to accept the sequence and a terminating null code element, which will be added automatically.

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

P Matches a set of code element sequences, which should be the same as the set of code element sequences that may be produced by the %P conversion of the **UniFprintf()** function. The corresponding argument shall be a pointer to a pointer to void. 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 behavior of the 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 code elements read from the input stream so far by this call to the **UniFscanf()** function. Execution of a %n directive does not increment the assignment count returned at the completion of execution of the function.

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

If a conversion specification is invalid, the behavior 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 characters matching the current directive (except for %n) 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 (if any) is terminated with an input failure.

If conversion terminates on a conflicting input, the offending input is left unread in the input stream. Trailing white space (including new−line 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.

The **UniFscanf()** function returns if an input failure occurs before any conversion. Otherwise, the 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.

23.3.4 UniSscanf() − Read a formatted string from memory

```
#include <UniChar.h>
int UniSscanf(
        const TextDataObject text_data_object,
        const UniChar *ucs,
        const UniChar *format,
        ...
        );
```

The **UniSscanf()** function is equivalent to the **UniFscanf()** function except that the **ucs** argument specifies a UniChar string from which the input is to be obtained, rather than from a stream. Reaching the end of the UniChar string is equivalent to encountering end−of−file for the **UniFscanf()** function. If copying takes place between objects that overlap, the behavior is undefined.

The **UniSscanf()** function returns −1 if an input failure occurs before any conversion. Otherwise, the **UniSscanf()** 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.

Glossary

This glossary includes the terms used in this document as well as other terms which may be of help to the user. Other sources of definitions for terms not found here are the IBM *Dictionary of Computing*, *SC20−1699*, ISO/IEC 10646, and Unicode 1.0.

Definitions identified with an asterisk (*), are taken from *The American National Dictionary for Information Processing*, copyright 1977 by the Computer and Business Equipment Manufacturers Association and obtainable through ANSI.

A

abstract character. A character as a semantic entity as opposed to a particular representation or shape of a character. See also character, 1.

accent mark. A maker placed above, below or beside a character to alter its phonetic value. See also diacritic.

ANSI. (1) American National Standards Institute. (2) The Microsoft Windows ANSI character set, equals to ISO 8859−1 with additional characters in the C1 area.

API. See *application programming interface*

application programming interface. A formally defined programming language interface between an IBM system control program or a licensed program, and the user of the program.

Arabic digits. Forms of decimal digits used in most parts of the Arabic world (for instance, U+0660, U+0661, U+0662, U+0663. Although European digits derive historically from these forms, they are visually distinct and coded separately. (Arabic digits are sometimes called Indic numerals; however, this leads to confusion with the digits currently used with the scripts of India.) Arabic digits are referred to as **Arabic−Indic digits** in the Unicode standard.

Arabic Numerals (also now known as European numerals in UCS). The numbers 0, 1, 2, 3, 4 5, 6, 7, 8, 9 as commonly used today.

ASCII(*). American Standard Code for Information Interchange. The standard code using a coded set consisting of 7−bit coded characters (8 bits including the parity bit), used for information interchange among data processing systems, data communication systems and associated equipment. The ASCII set consists of control characters and graphic characters. The US national variant of ISO 646. Formally, the U.S. standard ANSI X3.4−1986

attribute. A property or characteristic of one or more entities, for example color, weight.

B

backing store. Character storage in memory or on disk, as opposed to characters displayed or printed.

base character. (1) A character which is neither non−spacing nor characterized by obligatory overlapping with adjacent characters, nor by dependence upon other characters, that is one that stands on its own. (2) any graphic character which is not a non−spacing character. Base characters are also known as “spacing” characters, because they generally have non−zero width (in horizontal contexts).

Basic Multilingual Plane (BMP). Plane 0 of Group 0 in ISO 10646. See UCS−2.

BiDi. Acronym used for the bidirectional nature of Arabic and Hebrew text.

Big−endian. Refers to a computer architecture which stores multiple−byte numerical values with the most significant byte (MSB) values first.

Binary files. Files containing non−textual information.

bit. A binary digit, which can have a value of 0 or 1.

block. A contiguous collection of characters that share common characteristics, such as script. Within the Unicode encoding space, it is based on a multiple of 16. A typical Unicode block may contain unencoded positions, which are reserved.

BOM. Acronym for “Byte Order Mark”. The Unicode character (U+FEFF) used to indicate the byte order of a text. The BOM allows a receiver of Unicode text to distinguish between text arriving in big−endian order from text arriving in in litle−endian order *in the absence of a higher level protocol.*

byte. Data storage unit consisting of 8 bits, commonly used to represent a character in ASCII or EBCDIC.

C

canonical. (1) In Unicode, conforming to the general rules for encoding, that is, not compressed, compacted or in any other form specified by a higher protocol, and also not consisting of a Compatibility Zone encoding. (2) In ISO 10646, the form with which characters of this coded character set are specified utilizing four octets to represent each character.

CC−data−element.
(Coded−Character−Data−Element). An element of interchanged information that is specified to consist of a sequence of coded representations of characters, in accordance with one or more identified standards for coded characters ets.

CDRA. Character Data Representation Architecture.

cell. The place within a row of a code page at which an individual character may be allocated.

character. (1) The smallest component of written language that has semantic value. Character refers to the abstract idea, rather than a specific shape (see also glyph), though in code tables some form of visual representation is essential for the reader’s understanding. (2) The basic unit of encoding for the Unicode character encoding, 16 bits of information. (3) Synonym for ”code element”. (4) The English name for the ideograpic written elements of Chinese origin. (5) A member of a set of elements used for the organization, control, or representation of data.

character boundary. Within a stream of octets the demarcation between the last octet of the coded representation of a character

and the first octet of the next coded character.

Character Data Representation Architecture. An IBM architecture designed to provide a set of identifiers,

services, supporting resources and conventions in order to achieve consistent representation, processing and interchange of graphic character data in SAA environments.

character encoding. Association of a unique number with each character in a set of characters. The distinction between characters and glyphs is not absolutely clear in all cases, and so most large character encodings also encode some set of glyphs.

character set. A defined set of characters, with no underlying representation implied.

characteristics. Atrributes of a service which determine the functions and capabilities of the program while using the service.

CJK. Acronym for "Chinese, Japanese, and Korean."

code. A set of bit patterns, to which a specific control or graphic character meaning has been assigned.

code element. The minimal bit combitaion that can represent a unit of encoded text for processing or exchange.

code page. A specification of codepoints for each graphic character in a set or in a collection of graphic character sets. Within a code page, a code point may have only one unique specific meaning.

code point. A unique bit pattern defined in a code. Depending upon the code, a code point may be 7−bits, 8−bits, 16−bits or other bit combinations. Code points are assigned graphic or control characters in a code page.

code set. A character encoding; this term is widely used by programmers (see coded character set).

code table. A table showing the character allocated to the octets in a code.

coded character. A character together with its coded representation.

coded character set. A set of unambiguous rules that establishes a character set and the one−to−one relationship between the characters of the set and their coded representation.

codespace. The range of numerical values available for encoding characters, given the set of principles for how numerical values are to be used for encoding.

collate element. The smallest entity which may be used in collation operations to determine the logical order of a string (its collation sequence). A collation element may consist of either a single character or of two or more characters which will collate as a single entity.

collate sequence. A specified order used in sequencing. In the performance of collation some characters may be omitted and some characters may be given the same relative order (equivalence class).

combining character. A member of an identified subset of the coded character set of the ISO 10646 standard intended for combination with the preceding non−combining graphic character, or with a sequence of combining characters preceded by a non−combining character (See also *composite chracter sequence*).

common programming interface. Specification which provides languages, commands and calls that allow for the development of applications which can be more easily ported and integrated from system to system − a basic element of the IBM SAA.

compatibility character. A graphic character included as a coded character of ISO 10646 primarily for compatibility with existing coded character sets.

compatibility zone. A section of the Unicode codespace, included for compatibility with existing character encoding standards. The Compatibility Zone contains variant characters that can be mapped to Unicode canonical equivalents.

composite character sequence. A sequence of characters consisting of a base character followed by zero or more non−spacing marks.

conjunct consonant. (1) the juxtaposition of two or more consonants in Indic scripts. “Conjunt” refers to the pronunciation of two consonants with no intervening vowel. (2) The term is also used to refer to the specific typographic forms themselves, where the consonants may be written together in a single typographical form (ligature) or may be written with a **virama** below one or more of the consonants.

control function. An element of a character set that effects the recording, processing, transmission or interpretation of data, and that has a coded representation consisting of one or more octets.

CPI. See *common programming interface*.

cultural feature. An aspect that is expected within a culture to be done in a particular way, e.g. collate with culturally−expected results, or time and date formatting.

culture. A style of social expression particular to a society or class of a particular country using a particular language or combination of languages.

D

date type. A classification of data into different types, which have a determination in the handling of the data, examples are floating point, binary, integer and character.

DBCS. Double Byte Character Set (see also MBCS).

decomposition. A mapping a precomposed character into the corresponding composed character sequence.

default state. The state that is assumed when no state has been explicity specified.

detailed code table. An ISO 10646 code table showing the individual characters, and normally showing a single half−row.

device. A component of information processing equipment which can transmit and/or receive coded information as CC−data elements (it may be an input/output device in the conventional sense, or a process such as an application program or gateway function).

diacritic. Any character (spacing or non−spacing) used with the Latin script or related scripts, typically indicating that a phonetic value is different from the unmarked state. This broad definition is intended to include accents. See also *non−spacing diacritic*.

digits. See *Arabic digit, European digits, Indic digits*.

double byte character set. A coded character set in which each character is represented by a pair of bytes. A double byte coded character set is used to represent character sets which cannot be accommodated within the single byte encoding structure. Asian languages with their large character repertoires require the use of double byte character sets.

ductility. The ability of a cursive font to stretch or compress the connective baseline to effect text justitication.

E

EBCIDC. Extended Binary Coded Decimal Interchange Code, an encoding structure which allows for 190 graphic characters and 64 control characters, each character is composed of 8 bits.

enable (national language support). To design a product in a manner which will facilitate the inclusion of national language functions so that the product may function in international use. Such a product is enabled if (a) the national language version can be created efficiently, at minimal expense and (b) the product does not interfere with the current or planned national language support of other products.

encapsulated text. 1. Plan text surrounded by formatting information. 2. text re−coded to pass through narrow transmission channels or to match communication protocols.

endian−ness. General reference to *Big−endian* and *Little−endian*.

environment. A named collection of logical and physical resources used to support the performance of a function.

European digits. Forms of decimal digits used in Europe (for instance, 0, 1, 2, 3). Historically, these derive from the Arabic digits, but are visually distinct and coded separately. Many countries outside of Europe have also adopted these forms for decimal digits. (Although European digits are sometimes called “Arabic numerals,” this leads to confusion with the real Arabic digits).

EUC. Extended Unix Code, a character encoding which uses the ASCII code set and three additional supplementary code sets, to represent a character set.

F

fancy text. A data or interchange format containing both plain text and additional formatting information such as font, style, margins, etc.

FAT. File allocation table

file allocation table (FAT). In IBM personal computers, a table used by DOS to allocate space on a disk for a file and to locate and chain together parts of the file that may be scattered on different sectors so that the file can be used in random or sequential manner.

file code. The particular type of data encoding used in order to store data in a computer. File codes may be unique to the file or object managers, but may have a common data respresentation such as EBCDIC or ASCII.

floating. (~ diacritic. ~ accent, ~ mark) See *non−spacing mark*.

folding. The substitution of one character for another, where generally a larger character set is mapped to a

smaller set, which may result in the loss of information.

formatted text. See *fancy text*.

formatting codes. Non−spacing characters that are inherently invisible but which have an effect on the surrounding characters. An example is the `ZERO WIDTH NON-JOINER`.

G

GCGID. Acronym for “Graphic Character Global Identifier”. These are listed in the IBM document *Character Data Representation Architecture, Level 1, Registry SC09−1391*.

glyph. (1) The actual shape (bit pattern, outline) of a character image. Glyph is usually a synonym for image or character image. (2) a kind of idealized surface form derived from some combination of underlying characters in some specific context, rather than an actual character image. In this broad usage, two images would constitute the same glyph whenever they have essentially the same topology (as in oblique “**a**” and roman “**a**”), but different glyphs when one is written with a hooked top and the other without (as in italic ***a*** and roman a). In this usage “glyph” is a synonym for “glyph type, ” where glyph is defined as in sense 1.

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

graphic character set. A defined set of graphic characters, with no encoded representation.

graphic symbol. The visual representation of a graphic character.

group. A subdivision of the coding space of ISO 10646; of up to 256 x 256 x 256 cells.

H

Han. Generic adjective referring to ideographic characters of Chinese origin.

Han unification. Assignment of the same code point to characters from more than one of the East Asian ideographic character standards which they may (or may not) be represented by slightly different glyphs (in either of the two senses of “glyph” defined previously). This process actually reunifies elements which are historically used as, or have been historically perceived as being, the same “character.”

Hangul. The Korean symbol writing system.

Hanja. The Korean name for the ideographic characters of Chinese origin.

Hanzi. (**~ Han tsu**) The Chinese name for the ideographic characters of Chinese origin.

hard coded. Value coded as a constant, as opposed to a variable.

high performance file system (HPFS). In IBM personal computers, a new file system introduced in OS/2. It is an alternative to the FAT file system that is also supported by OS/2.

HINDI numerals. The presentation shape of numerals as used in parts of the Arabic world to represent the numbers 0 through 9.

HLL. High Level Language − one of the progamming languages such as Cobol, PL/1, Fortran, Pascal, etc. (not assembler level).

HPFS. High performance file system.

I

Ideographic character. A character that generally stands for a word or a morpheme, rather than a sound. In this context the term is applied to characters of the Chinese script used for writing various languages. The Chinese script can be more accurately described as “logographic” but the Unicode standard uses the more widely−known terminology. The term “ideograpic” is often also applied to various hieroglyphic writing systems.

IEC. International Electrotechnical Commission.

IEEE. Institute of Electrical and Electronics Engineers.

IEEE Standard 1003.2−1992. Part of the POSIX series of standards for applications and user interfaces to open systems. It defines the applications interface to shell command language, and a set of utility programs for complex data manipulation.

implement (national language support).. The provision of the specific values needed by the enabled functions of a product in order for that product to support a specific user’s language, culture and encoding needs.

indic digits. Forms of decimal digits used in various Indic scripts (for instance, Devanagai: U+0966, U+0967, U+0968, U−0960). Although Arabic digits (and eventually, European digits) derive historically from these forms, they are visually distinct and coded separately.

interchange. The transfer of character coded data from one user to another.

interworking. The process of permitting two or more systems, each employing different coded character sets, meaningfully to interchange character coded data; conversion between the two codes may be involved.

invariant character set. A character set such as the syntactic character set whose code point assigments do not change from one code page to another (within the same encoding structure). A minimal set of characters which is available as a common part of all character sets.

ISCII. (1) Indian Standard Code for Information Interchange. (2) Iranian Standard Code for Information Interchange.

ISO. International Organization for Standardization.

ISO/IEC 9945−2: 199x. ISO/IEC equivalent of the IEEE Standard 1003.2−1992 (POSIX.2).

K

Kanji. The Japanese name for the ideographic characters of Chinese origin.

keyboard layout. The designation of which keys produce which characters (or scan−codes).

L

LANG. An environment variable used to set the required language, territory and code set in a POSIX−conforming system. The value of LANG may be used to initialize the program locale (same as the use of category LC_ALL). LANG actually contains the name of the locale which is currently in effect.

language. Set of characters, conventions, rules for conveyance of information which includes syntax and semantics. Should be specified as to whether it is a Natural Language or a Programming Language.

LC_ALL. A category used by the function setlocale to name the program's entire locale. Valid categories are defined for this function, see the following LC_terms.

LC_COLLATE. An environment variable used set the user's requirements for the character collation sequence, and a category used by setlocale the function to name the character collation information to be used in a locale.

LC_CTYPE. An environment variable used to set the user's requirements for character classification and case conversion, and a category used by the setlocale function to identify the character type and character shift information in a locale.

LC_MONETARY. An environment variable used to set the user's requirements for currency and monetary value format, and a category used by the setlocale function to name the monetary formatting information to be used in a locale.

LC_NUMERIC. an environment variable used to set the user's requirements for the decimal number format, and a category used by the setlocale function to identify the number formatting information in a locale.

letter. (1) Basic element of script as understood by the end user. (2) A higher level of abstraction than "character". See *text element*.

library. A file or set of related files.

library routine. A proven routine maintained in a program library.

ligature. Two (or more) characters combined into a single typographical form. In the Latin script, there are only a few in modern use, such as the ligatures between "f" and "i" (=fi) or "f" and "l" (=fl). Other scripts make use of many ligatures, depending on the font and style. Some languages have mandatory ligatures; other languages have text elements that historically were derived from ligatures, but are now characters. Examples of the latter are the German **eszet** (the ligature of long and short "s") and the ampersand (&) which originated as a contracted form of the Latin word "et" (which derivation can still be discerned in many fonts).

Little−endian. Refers to a computer architecture which stores multiple−byte numerical values with the least significant byte (LSB) values first.

locale. (1) Geographic − a region, area, nation which shares language, customs, alphabets and symbols, currency, calendar etc. (2) Computer − refers to collection of related processing variables, which are used to define the localized behavior of a process during its execution. For NLS, local restricts itself to only those variable which pertain to the language, customs and coded character sets of the

user. In the POSIX conforming systems, the selection of the LANG environment variable will provide the default setting for the locale. (3) A subset of the user's environment which defines the conventions for a specific language or culture (for multilingual use, it will also have a specification for a number of languages and cultures), such as date and time formatting, character conversion, classification and collation.

localedef. A utility used to convert the source form of a locale into a form which can be used by functions and utiities.

localization. Adaption or implementation of software to meet requirements of a particular group of users, usually based on country or local environment, which will require the sensitivity to language, customs and coded character sets. A program which requires only translation of the program's messages is said to be locale independent; the operating system provides the parameters for the program to reconfigure itself. The usual examples are different date and time formatting.

logographic. See Ideographic Character; Chinese characters are more properly termed logographic than ideographic.

LSB. Acronym for **least significant byte**.

LTR. Left−To−Right orientation of text.

M

MBCS. Acronym for *multibyte* character set. This implies one or more bytes per character and often, that the number of bytes may be different for different characters in the character set. Many large character sets have been defined as MBCS in order to keep strict compatibility with ASCII subset and/or ISO 2022.

monocase. Transformation of characters from one case to other, as in Latin alphabetic characters from lower case to the corresponding character in upper case (not
valid in all languages or writing systems).

MSB. Acronym for **most significant byte**.

multibyte. The term used for character encodings that employ one or more bytes per character; some such encodings may have a variable number of bytes per character. These encodings typically follow the ISO 2022 model, avoiding use of the byte values in the so−called control ranges to encode graphic characters.

multilingual. Referring to multiple languages. A multilingual program strives to handle data in a way that is independent of language or writing system. Multilingual documents combine text which is written in different languages. Multilingual may refer to languages which use the same script (such as English, French, and German), or to languages which use distinct scripts (such as German, Hebrew, and Korean). The latter case is also referred to as **multiscript.**

N

national convention. A convention which is specific to a particular nation or district, and which may refer to everything from the currency symbol to the names of the weekdays in any language or local dialect. Somewhat synonymous with "locale", but the emphasis is not on the computer representation.

national language. First or primary language of a person or user.

national language support. (1) Model of support for users in order to provide a machine interface which

respects the national language requirements of the user (2) Ability to allow users to communicate with the product in the language most familiar to the user.

neutral character. A character which can appear within either right−to−left or left−to−right text.

non−spacing character. A character with character width (advance width) equal to zero. There are two types of non−spacing characters: formatting codes and non−spacing marks. See *formatting codes* and *non−spacing mark marks*.

non−spacing diacritic. A diacritic which is a non−spacing mark.

non−spacing mark. A non−spacing graphic character which is positioned with reference to a preceding base character.

O

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

open system. A system with specified standards which can be readily connected to another system which complies with the same standards.

P

plain text. Computer encoded text which consists only of a sequence of code elements from a given standard, with no other formatting or structural information to indicate a specific interpretation. Plain text interchange is commonly used between computer systems which may have no other factors in common.

plane. In ISO 10646, a subdivision of a group; of up to 256 x 256 cells.

points. The non−spacing vowels and other signs of written Hebrew.

POSIX. Portable Operating System Interface.

precomposed character. A single Unicode character which represents a composed character sequence, usually a combination of one or more diacritic marks with a base character.

presentation. The process of printing, or displaying a graphic symbol.

presentation form. In the presentation of some scripts, a form of a graphic symbol representing a character that depends on the position of the character relative to other characters.

process code. The particular type of
encoding used by a computer when processing information. The process code may be the same as or different from the encoding used to store date in files.

profile. Data that describes the significant charactristics of a user, group of users, or one or more computer resources.

profile file. A file which contains commands to be executed at logon time and variable profile assignments

to be set or initialized. A system profile would contain the values to be used by all users, whereas the individual user profile would contain information unique to a specific user. The individual user profile values, when processed, would overide those of the system supplied profile.

R

radical. Parts of Chinese characters used collectively as a method of indexing them A character may contain more than one element which is recognized as being a Radical, but each contains only one element that is actually used as the indexing Radical for the character. Many radicals can exist as stand−alone Characters.

radix point. Location of the separation point between the characters associated with the integral part and those associated with the fractional part − in North American use
the (.) period is used to designate this point.

rendering. The text process of displaying or printing a sequence of characters in a visible format.

replacement character. Character used to substitute for an uninterpretable character from another encoding. The Unicode standard uses U+FFFD (REPLACEMENT CHARACTER) for this function.

replacement glyph. (Synonym: missing glyph) a glyph used to render a character which cannot be rendered with the correct appearance in a particular font. It often is shown as an open box or as a black rectangle.

row. In ISO 10646, a subdivision of a plane; of up to 256 cells.

RTL. Right−to−Left orientation of text.

S

SAA. See *Systems Application Architecture*.

SBCS. See *single byte character set*.

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

session. Uninterrupted time between logon/login and logoff.

setlocale. A function which is used to change or query a locale. It has two arguments, the first is the category and the second is the locale. The locale argument is used to provide settings for the category specified.

single byte character set. A coded character set in which each character is represented by a single byte code point.

SJIS. Shift−Japanese Industrial Standard, an encoding scheme consisting of single bytes and double bytes used for character encoding. Because of the large number of characters in the Japanese and other Asian languages, the 8−bit byte is not sufficient for character encoding.

sort. To arrange a set of items according to keys used as the basis for determining the sequence of the items.

string. Connected group of characters.

supplementary planes. In ISO 10646, planes that accommodate characters which have not been allocated to the Basic Multilingual Plane.

supported language. Language for which various elements/attributes have been implemented on the particular system.

syntactic character set. A set of 81 graphic characters registered in the IBM registry as CS 640. This set was originally recommended to the programming languages to be used for syntactic purposes and to achieve portability and interchangeability across systems and country boundaries.

Systems Application Architecture. A set of IBM defined software interfaces, conventions and protocols which provides a framework for the design and development of applications with cross systems consistency as an objective.

T

text element. A minimal unit of text relative to some specific language and some specific process performed on the text.

U

UBA. The Unicode Bidirectional Algorithm. It is defined in Unicode standard and specifies the rendering of a **visual** Unicode string representation from its **logical** equivalent used to keep Bidi text in computer backing store.

UCS−2. Universal Coded Character Set, consisting of two−byte code elements. Equivalent to ISO 10646 BMP and Unicode.

UCS−4. Universal Coded Character Set, consisting of four−byte code elements. Equivalent to the entire ISO 10646 code page.

unification. The folding together of elements that need not be distinguished, or the recognition that two things which are distinguished in some places should not, in
general, be distinguished as encoded characters because their use is disjoint enough to be apparent from context. Two ends of the spectrum are “radical” unification, which might fold together the upper and lowercase forms of Greek A, Cyrillic A and Latin A into a single code point; and the more conservative unification of the Unicode standard, which maintains boundaries between scripts in the broadest sense and, for compatibiity, preserves many distinctions made in other computer character encodings.

Universal Coded Character Set (UCS). The ISO 10646 standard which defines a single data encoding scheme for textual information in the data processing environment.

user. (1) Anyone who requires the use of services of a computing system. (2) A person or other entity that invokes the
service provided by a device. (This entity may be a process such as an application program if the “device” is

a code converter or a gateway function, for example).

user profile. A description of a user that includes information such as USER ID, password, access authority and other atttributes obtainable at logon.

UTF−1. UCS Transformation Format is specified in Annex G of ISO 10646. It can be used to transmit text data through communication systems which are sensitive to octet values for coded control characters according to the structure of ISO 2022.

V

value. A specific occurrence of an attribute.

variable. Data storage location used in programming which may take on different values, one at a time.

vowel sign. A mark used in many scripts to indicate a vowel or vowel quality Typical vowel signs are either partially overlapping or non−spacing marks, but those are not necessary criteria. Vowel signs are characterized generally by **functional** dependence upon an adjacent or nearby consonant, the inherent vowel of which they typically modify.

W

wchar_t. The ANSI C defined **wide character** type, usually implemented as either 16 or 32 bits. ANSI specifies that wchar_t be an integral type and that the C language source character set be mappable by simple extension (zero− or sign− extension). A frequent assumption is that the source and target code sets are different, and that the size of the “char” data type is insufficiently “wide” to hold a character of the target code set.

Z

zero width. See *non−spacing mark.*

zone. A sequence of cells of a code table, comprising one or more rows, either in whole or in part, containing characters of a particular class.

Bibliography

[1] *ISO/IEC 10646−1:1993*, Information Processing − Universal Coded Character Set (UCS)

[2] *ISO/IEC 9945−2*:...., Information technology − Portable operating system interface (POSIX)−Part 2: Shell and Utilities

[3] *ISO DIS 9541, 1990−02−06.* Information Processing − Font Information Interchange.

[4] *POSIX 1003.1−1988.* ANSI/IEEE Standard Portable Operating System Interface for Computer Environments, ISBN 1−55937−003−3.

[5] *SAA Common Programming Interface C Reference − Level 2,* SC09−1308−02

[6] *Character Data Representation Architecture − Level 1, Reference,* SC09−1390−00

[7] *National Language Information and Design Guide.* SE09−8000.

[8] *National Language Design Guide*, Volume 1, Designing Enabled Products, Rules and Guidelines, SE09−8001

[9] *National Language Design Guide*, Volume 2, Left−to−Right Languages and Double−Byte Character Set Languages, SE09−8002

[10] *Internationalization of AIX Software − A Programmer's Guide*, IBM, 1992, SC23−2431−00

[11] *Font Object Content Architecture***.** S544−3285

[12] *SHARE Europe (SEAS) White Paper* **(Ed. Klaus Daube).** National Language Architecture, Pub. SHARE Europe, 17 Rue Pierres−du−Niton, CH−1207 Geneva, Switzerland.

[13] *The Unicode Standard, Worldwide Character Encoding*, Version 1.0, Volume 1, Addison−Wesley Publishing Company, Inc. 1991, ISBN 0−201−56788−1

[14] *The Unicode Standard, Worldwide Character Encoding*, Version 1.0, Volume 2, Addison−Wesley Publishing Company, Inc., 1992, ISBN 0−201−60845−6

[15] **Becker, Joseph D (1984).** *Multilingual Word Processing*. Scientific American 251(1):96−107.

[16] **Clews, John.** Language Automation Worldwide − The development of character set standards. Pub. Sesame Compute Projects, 8 Avenue Road, Harrogate, HG2 7PG, UK. ISBN 1−870095−01−4.

[17] **Comrie, B (1987). Ed..** *The World's Major Languages*, Croom Helm Ltd..

[18] **Comrie, B (1981).** *The Languages of the Soviet Union*, Cambridge University Press.

[19] **Coulmas, F (1989).** *The Writing Systems of the World*, Basic Blackwell Ltd.

[20] **Gelb, IJ (1963).** *A Study of Writing*, Chicago, London: University of Chicago Press (revised edition, first published 1952).

[21] **Katzner, K (1986).** *The Languages of the World*, Routledge & Keegan Paul (revised edition, first published 1977).

[22] **Sampson, GR (1985).** *Writing Systems*, Stanford University Press.

[23] **Sibley, E. H. et al.** *Alphabets and Languages*, Communications of the ACM, May 1990, Vol 33 number 5.

[24] **Simons, Gary F.** "The Computational Complexity of Writing Systems", *The Fifteenth LACUS Forum* 1988, Linguistic Association of Canada and the United States, Lake Bluff, IL (1988).

[25] **Shrestha, Vinod P et al..** Japanese Text Formatting, IBM research Technical Report RT 5001, February 3rd, 1989.

[26] **Suignard, M (1989).** NLS Bundle, Microsoft paper (unpublished).

[27] **X/Open Company Ltd.**, *X/Open CAE Specification System Interface Definitions, Issue 4*, X/Open Company Ltd., 1992, ISBN:1−872630−46−4

[28] **X/Open Company Ltd.**, *X/Open CAE Specification Commands and Utilities, Issue 4*, X/Open Company Ltd., 1992, ISBN:1−872630−48−0

[29] **X/Open Company Ltd.**, *X/Open CAE Specification System Interfaces and Headers, Issue 4*, X/Open Company Ltd., 1992, ISBN:1−872630−47−2