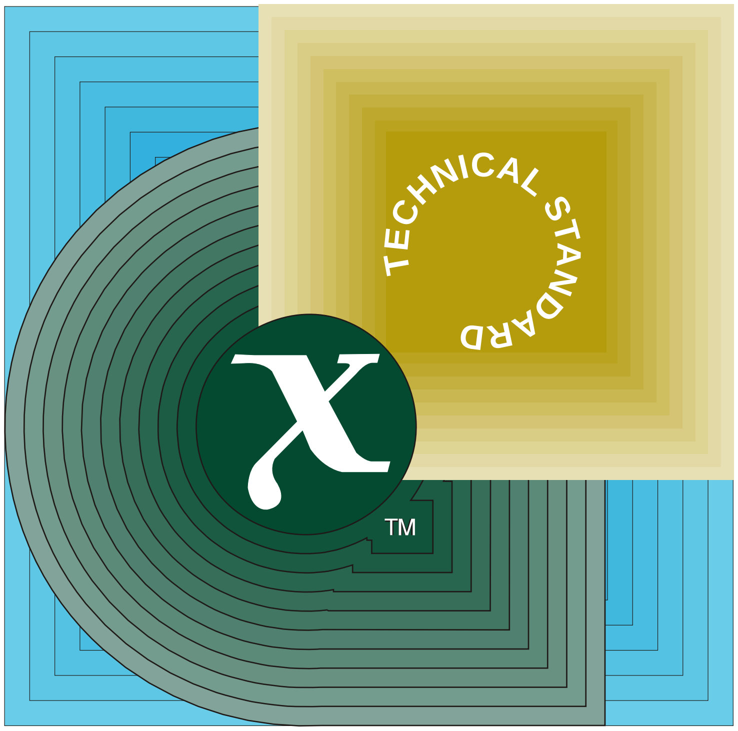
Technical Standard

Portable Layout Services: Context-Dependent and Directional Text



THE Open GROUP

[This page intentionally left blank]

CAE Specification

Portable Layout Services:

Context-dependent and Directional Text

The Open Group

CAE Specification

Portable Layout Services: Context-dependent and Directional Text

ISBN: 1-85912-142-X
Document Number: C616

Published in the U.K. by The Open Group, February 1997.

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

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

or by Electronic Mail to:

OGSpecs@opengroup.org

Contents

List of Examples

List of Figures

List of Tables

Preface

The Open Group

The Open Group is an international open systems organisation that is leading the way in creating the infrastructure needed for the development of network-centric computing and the information superhighway. Formed in 1996 by the merger of the X/Open Company and the Open Software Foundation, The Open Group is supported by most of the world's largest user organisations, information systems vendors and software suppliers. By combining the strengths of open systems specifications and a proven branding scheme with collaborative technology development and advanced research, The Open Group is well positioned to assist user organisations, vendors and suppliers in the development and implementation of products supporting the adoption and proliferation of open systems.

With more than 300 member companies, The Open Group helps the IT industry to advance technologically while managing the change caused by innovation. It does this by:

- consolidating, prioritising and communicating customer requirements to vendors
- conducting research and development with industry, academia and government agencies to deliver innovation and economy through projects associated with its Research Institute
- managing cost-effective development efforts that accelerate consistent multi-vendor deployment of technology in response to customer requirements
- adopting, integrating and publishing industry standard specifications that provide an essential set of blueprints for building open information systems and integrating new technology as it becomes available
- licensing and promoting the X/Open brand that designates vendor products which conform to X/Open Product Standards
- promoting the benefits of open systems to customers, vendors and the public.

The Open Group operates in all phases of the open systems technology lifecycle including innovation, market adoption, product development and proliferation. Presently, it focuses on seven strategic areas: open systems application platform development, architecture, distributed systems management, interoperability, distributed computing environment, security, and the information superhighway. The Open Group is also responsible for the management of the UNIX trade mark on behalf of the industry.

The X/Open Process

This description is used to cover the whole Process developed and evolved by X/Open. It includes the identification of requirements for open systems, development of CAE and Preliminary Specifications through an industry consensus review and adoption procedure (in parallel with formal standards work), and the development of tests and conformance criteria.

This leads to the preparation of a Product Standard which is the name used for the documentation that records the conformance requirements (and other information) to which a vendor may register a product. There are currently two forms of Product Standard, namely the Profile Definition and the Component Definition, although these will eventually be merged into one.

The X/Open brand logo is used by vendors to demonstrate that their products conform to the relevant Product Standard. By use of the X/Open brand they guarantee, through the X/Open Trade Mark Licence Agreement (TMLA), to maintain their products in conformance with the Product Standard so that the product works, will continue to work, and that any problems will be fixed by the vendor.

Open Group Publications

The Open Group publishes a wide range of technical literature, the main part of which is focused on specification development and product documentation, but which also includes Guides, Snapshots, Technical Studies, Branding and Testing documentation, industry surveys and business titles.

There are several types of specification:

- *CAE Specifications*

  CAE (Common Applications Environment) Specifications are the stable specifications that form the basis for our product standards, which are used to develop X/Open branded systems. These specifications are intended to be used widely within the industry for product development and procurement purposes.

  Anyone developing products that implement a CAE Specification can enjoy the benefits of a single, widely supported industry standard. In addition, they can demonstrate product compliance through the X/Open brand. CAE Specifications are published as soon as they are developed, so enabling vendors to proceed with development of conformant products without delay.

- *Preliminary Specifications*

  Preliminary Specifications usually address an emerging area of technology and consequently are not yet supported by multiple sources of stable conformant implementations. They are published for the purpose of validation through implementation of products. A Preliminary Specification is not a draft specification; rather, it is as stable as can be achieved, through applying The Open Group's rigorous development and review procedures.

  Preliminary Specifications are analogous to the *trial-use* standards issued by formal standards organisations, and developers are encouraged to develop products on the basis of them. However, experience through implementation work may result in significant (possibly upwardly incompatible) changes before its progression to becoming a CAE Specification. While the intent is to progress Preliminary Specifications to corresponding CAE Specifications, the ability to do so depends on consensus among Open Group members.

- *Consortium and Technology Specifications*

  The Open Group publishes specifications on behalf of industry consortia. For example, it publishes the NMF SPIRIT procurement specifications on behalf of the Network Management Forum. It also publishes Technology Specifications relating to OSF/1, DCE, OSF/Motif and CDE.

  Technology Specifications (formerly AES Specifications) are often candidates for consensus review, and may be adopted as CAE Specifications, in which case the relevant Technology Specification is superseded by a CAE Specification.

In addition, The Open Group publishes:

- *Product Documentation*

  This includes product documentation — programmer's guides, user manuals, and so on — relating to the Pre-structured Technology Projects (PSTs), such as DCE and CDE. It also includes the Single UNIX Documentation, designed for use as common product documentation for the whole industry.

- *Guides*

  These provide information that is useful in the evaluation, procurement, development or management of open systems, particularly those that relate to the CAE Specifications. The Open Group Guides are advisory, not normative, and should not be referenced for purposes of specifying or claiming conformance to a Product Standard.

- *Technical Studies*

  Technical Studies present results of analyses performed on subjects of interest in areas relevant to The Open Group's Technical Programme. They are intended to communicate the findings to the outside world so as to stimulate discussion and activity in other bodies and the industry in general.

- *Snapshots*

  These provide a mechanism to disseminate information on its current direction and thinking, in advance of possible development of a Specification, Guide or Technical Study. The intention is to stimulate industry debate and prototyping, and solicit feedback. A Snapshot represents the interim results of a technical activity.

Versions and Issues of Specifications

As with all *live* documents, CAE Specifications require revision to align with new developments and associated international standards. To distinguish between revised specifications which are fully backwards compatible and those which are not:

- A new *Version* indicates there is no change to the definitive information contained in the previous publication of that title, but additions/extensions are included. As such, it *replaces* the previous publication.
- A new *Issue* indicates there is substantive change to the definitive information contained in the previous publication of that title, and there may also be additions/extensions. As such, both previous and new documents are maintained as current publications.

Corrigenda

Readers should note that Corrigenda may apply to any publication. Corrigenda information is published on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

Ordering Information

Full catalogue and ordering information on all Open Group publications is available on the World-Wide Web at **http://www.opengroup.org/public/pubs**.

This Document

This document is a CAE Specification (see above). It describes a set of portable functions for handling context-dependent and bidirectional text transformations as a logical extension to the existing POSIX locale model.

Structure

This document is structured as follows:

- Chapter 1 is an introduction.
- Chapter 2 is an overview of aspects of the writing systems of a large family of languages that are collectively called complex-text languages, including bidirectional languages. If you are familiar with complex-text languages and the problems associated with their processing and presentation, you need not read this chapter.
- Chapter 3 outlines the purpose of the *m_*_layout*( ) functions, which facilitate the transformation of data from one form to another.
- Chapter 4 describes the data types and layout values that are defined in the header file **<sys/layout.h>**.
- Chapter 5 contains reference manual pages for the *m_*_layout*( ) functions in alphabetical order.
- Appendix A is an implementation example.
- Appendix B describes the LO_LTYPE category, which is basically an extension to the existing LC_CTYPE category.
- Appendix C defines the interface between the locale-independent and locale-dependent layer of Layout Services.

Intended Audience

This document is intended for system and application programmers who want to provide support for complex-text languages.

Typographical Conventions

The following typographical conventions are used throughout this document:

- **Bold** font is used in text for filenames, keywords, type names, data structures and their members.
- *Italic* strings are used for emphasis or to identify the first instance of a word requiring definition. Italics in text also denote:
  - — variable names, for example, substitutable argument prototypes
  - — environment variables, which are also shown in capitals
  - — external variables, such as *errno*
  - — functions; these are shown as follows: *name*( )
- Normal font is used for the names of constants and literals.
- The notation **<file.h>** indicates a header file.
- Names surrounded by braces, for example, {ARG_MAX}, represent symbolic limits or configuration values which may be declared in appropriate headers by means of the C

#define construct.

- The notation [EABCD] is used to identify an error value EABCD.
- Syntax, code examples and user input in interactive examples are shown in `fixed width` font.
- **`Bold fixed width`** font is used to identify brackets that surround optional items in syntax, **`[ ]`**, and to identify system output in interactive examples.
- Variables within syntax statements are shown in *`italic fixed width font`*.
- Shading is used as described in Section 1.4 on page 3.

Trade Marks

Acknowledgements

The Open Group gratefully acknowledges the work of the authors of this document:

Israel Gidali (IBM Canada)
John Gioia (IBM Canada)
Frank Rojas (IBM Austin)

In addition many members of the Joint X/Open UniForum Internationalisation Group (XoJIG) have contributed by reviewing earlier drafts.

This document could not have been prepared without the considerable contribution of many. The functions defined for Complex-text Language Layout transformations can be seen as the product of the effort of the architects and developers of the traditional Bidirectional support and the emerging Unicode focus on directionality.

Special thanks go to Isai Scheinberg, Milos Lalovic and Mel Sher (IBM Canada), and to Mati Allouche (IBM Israel), for their contribution to the initial specifications of the ‘‘BIDI_HANDLE’’, the predecessor of **LayoutObject**. Their work laid the foundation for the Layout Values that are needed across all locales.

Khaled Sherif (IBM Egypt) and Ehud Nathan (IBM Israel) take most of the credit for the actual final specifications based on their BIDI support on AIX. Their insight and experience in development of BIDI software is the foundation for much of the specification. Much of the prototype and feasibility has been done by them and their respective teams in their countries.

The team led by Nelson Ng (SunSoft) and consisting of Ienup Sung (SunSoft), Shinobu Matsuzuka (SunSoft), Randy Hill (Sun) and Chookij Vanatham (CDG Systems Ltd.), is to be credited for the prototype work on the first Thai implementation of the Layout Services. The Thai prototype has provided valuable feedback regarding the implementation feasibility of the composite character support. Nelson Ng is to be credited also for the appendix describing the Dynamic Pluggable Interface of locale-specific Layout Services.

Frank Rojas is to be credited for his creative consolidation of previously divergent and fragmented ideas and for the foundation of the architectural concepts of Layout Object as published in a document called The LAYOUT Utility from which the suggested APIs and the **LayoutObject** originate.

In preparing the Layout Services sections, the complex-text languages overview and the LO_LTYPE, and in the editing of this proposal, Israel Gidali has relied upon information originating from a multitude of sources. Thanks go to:

Apisak Apiwathanokul, Ranat Thopunya and Jenwit Sriwiwattanangkoon (IBM Thailand) whose work was the main source used by the editor to compile the Thai Language section in the overview.

Seong Mook Kim (IBM Korea) for his enlightening information regarding the Korean writing system. Alexis Cheng (NLTC, IBM Canada) for his help with the syntax aspects of the modifiers.

Mike Feldman (Digital Equipment Corporation Israel), Nelson Ng (Sun), Ienup Sung (Sun), Ranya Abdel Rahman (IBM Egypt), Chun-Wan Lai (Digital Equipment Corporation Hong Kong), Chookij Vanatham (CDG Systems Limited), Eun Yoon Young (IBM Korea), Lisa Moore (IBM USA), Anuwat Phrukphicharn (HP Thailand) and Judy Chen (HP USA) for their very constructive suggestions.

John Gioia (IBM Canada) who has helped the editor through the esoteric alleys of UNIX and the C language and for his contributions to the clarification of the XPG implications and the preparation of the graphic images. Pini Schapira and Adiva Gera (IBM Israel) who started the initial effort to propose standard enhancements to accommodate the bidirectional transformations.

Zarko Cvijan, Dr. Hussein Kushki, Fred Bealle, Dr. Umamaheswaran V.S., Sheila Richardson, Anne Stilman, and Willy Rose from NLTC (IBM Canada), Taweesak Kositapan (IBM Thailand), Alaa Eddine M. Ghoneim, and Sherif El-Rafei (IBM Egypt), Mati Allouche and Ari Erev (IBM Israel) for their review and valuable comments on the drafts of the different components of the proposal.

And last, but not least, Carol Painting, former X/Open editor, who has done an excellent job in compiling, restructuring, editing, simplifying and moulding this document.

Referenced Documents

The following documents are referenced in this specification:

Distributed Internationalisation Services

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

ECMA TR / 53

European Computer Manufacturers Association, Handling of Bidirectional Texts, 2nd Edition — June 1992.

ISO / IEC 6429

ISO / IEC 6429: 1992, Information Technology — Control Functions for Coded Character Sets.

ISO / IEC 10646

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

Unicode

The Unicode Standard, Version 2.0, Addison-Wesley Publishing Company, Inc.1996.

wtt2.0

National Electronics and Computer Technology Center (NECTEC), Computer with WTT Thai Language Draft Proposal (ISBN 974-7570-66-1).

X11R5 Xlib

CAE Specification, May 1995, Window Management (X11R5): X Lib - C Language Binding (ISBN: 1-85912-088-1, C508).

XSH, Issue 5

CAE Specification, January 1997, System Interfaces and Headers, Issue 5 (ISBN: 1-85912-181-0, C606).

Chapter 1

Introduction

This chapter covers the scope of this document and introduces several important concepts. For definitions of unfamiliar terms, refer to the glossary.

1.1 Scope

This document describes a set of portable functions for handling context-dependent and bidirectional text transformations. It is a logical extension to the existing POSIX locale model.

Chapter 2 provides a detailed overview of the writing system aspects of complex-text languages, including the *bidirectional languages*,[1] Korean and Thai. If you are familiar with complex-text languages and the problems associated with their processing and presentation, you need not read Chapter 2.

This document introduces the concepts of *complex-text languages* and *layout transformations*. It defines a locale-associated object called **LayoutObject** to handle layout transformations. The layout object may contain an additional locale category, LO_LTYPE, containing character classifications and mappings for layout transformations.

This document also defines a standard set of *layout values* containing layout text descriptors and layout action indicators, and a set of Application Programming Interfaces (APIs) to facilitate layout transformations for complex-text languages.

This document addresses the need to enhance the existing XPG locale model. It provides support for the transformations between the layout of text during processing and the layout of text during presentation of complex-text languages, as an addition to the existing support for the basic locale services (parsing, tokenising, and so on).

Though the proposed concepts are not constrained to systems with a C-language implementation only, C-language bindings are provided for the APIs, and C-language examples are used.

1. The term *bidirectional languages* is used throughout this document instead of ‘‘languages with a bidirectional script’’. The latter is the correct form, but is more cumbersome.

1.2 Example

OP An example of the need for layout transformations is the implementation of the support for bidirectional text in Motif. This implementation involves transformations of a *logical stream* of bidirectional text into a *physical stream* presented on a display, with *segment* inversions and some character *shaping* using an intermediary *explicit type of text* (see Figure 1-1). For a detailed explanation of the different terms used here, see Chapter 2.

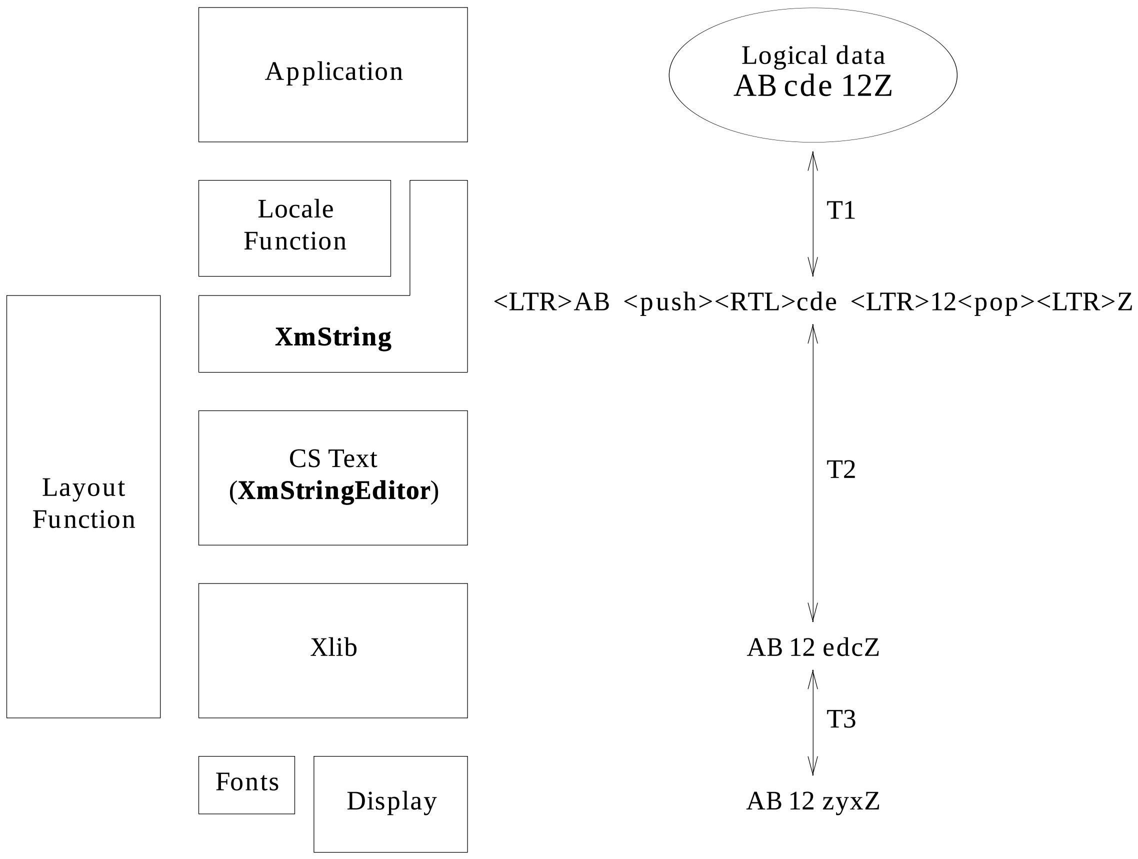


Figure 1-1 Layout Transformations in Motif

OP The transformations transform the logical data to an internal explicit layout (T1), and then prepare it for presentation by inverting directional segments (T2) and shaping the national characters, symbolically represented here by lower-case letters (T3).

1.3 Target Audience

This extension benefits:

- developers of operating system services or toolkits (such as Motif, terminal emulators, printer filters) that want to provide support for complex-text layout transformation functions in a standard, portable way
- developers of application programs, using presentation services or toolkits, in an environment that supports the layout transformation functions, as follows:
  - — to modify the system defaults of layout values, when using the layout services provided by the operating system
  - — to use the layout functions for transforming a complex text that has a peculiar layout
  - — to have direct control on the complex-text language presented on a screen or on a printer
- application developers in an environment that does not provide layout transformation services for complex-text languages, who are willing to implement the proposed APIs.

Existing applications, using internationalised functions (ISO C, XPG4, POSIX, Motif and X), that are performing logical processing without the need for layout transformations, may not be affected by this extension. The vast majority of applications that use the standard presentation services (Motif, curses, and so on) or the layout presentation services that may be provided by the operating system, with the standard defaults, should not be affected.

1.4 Approach

The set of layout functions address transformation between the layouts of text during processing and the layouts of text during presentation with appropriate APIs. The associated standard set of character classifications, mapping and text layout attributes are defined; the framework for grouping, storing and modifying these attributes is also defined.

Possible implementation examples, as well as some components of the Layout Services (such as some specific values of the layout values), have been marked in the document by shading and the letters ‘‘OP’’ in the margin to indicate that they are not mandatory.

Chapter 2

Complex-Text Languages — An Overview

This chapter contains text which is reprinted with the permission of the author, Israel Gidali.

This chapter presents an overview of aspects of the writing systems of a large family of languages that are collectively called *complex-text languages*. Section 2.1 is an introduction to complex-text languages. Section 2.2 on page 7 discusses bidirectional languages, in particular Arabic and Hebrew, including their use in a data processing environment. Section 2.4 on page 18 and Section 2.5 on page 23 describe other typical complex-text languages (Thai and Korean). Section 2.6 on page 26 summarises the main principles and suggests a few guidelines for application developers.

2.1 Complex-text Languages

In the languages of the western world based on the Latin, Cyrillic and Greek scripts, there is no difference between how text is stored for data processing and how it is presented on a display or a printer. The text is read on horizontal lines from left to right, the lines progress from top to bottom and the characters are stored in a manner identical to how they are presented.

Not all the languages of the world have these characteristics.

In this document, *complex-text languages* are defined as those languages for which the text has a different layout when presented from when it is stored for data processing. The term *layout*, which is equivalent, in this context, to the term *format*, refers to the shape of the characters and the direction of portions of the text.

An additional characteristic of complex-text languages (with the exception of Vietnamese) is the fact that they do not have upper-case or lower-case characters.

Typical complex-text languages are those with a *bidirectional script*. Usually they are written from right to left, with some portions of text, such as numbers and embedded Latin-based text, written from left to right. Bidirectional languages include the languages of the Middle East and Africa (Arabic, Hebrew, Urdu, Farsi, Yiddish, and so on). Other complex-text languages include some languages of Asia that do not limit their encoding to a double-byte scheme (Thai, Lao, Vietnamese, Korean, and so on).

There is nothing in these languages themselves that is more complex than in the Latin-based languages; they are special only in that the presented text does not necessarily look identical to the text as stored.

Though the term *complex* is used to describe the text of the bidirectional and some other Asian languages, enabling a program to work in these languages is relatively simple, once the peculiarities of these languages are understood.

2.1.1 Layout Transformations and Related Attributes

To enter, process and present a text in a complex-text language, it is necessary to perform *transformations* between the *processing layouts* and the *presentation layouts*. The processing layout is the layout of text when stored or processed. The presentation layout is the layout of text when presented on a display or a printer.

These transformations have to take into account specific *text attributes*, including directionality, shaping, composition of characters and national numbers. Text attributes that describe bidirectional writing systems are defined in Section 2.2 on page 7.

An internationalised application must be designed to deal automatically with this kind of transformation and related attributes.

2.2 Bidirectional Languages

The bidirectional languages are used mainly in the Middle East. They include Arabic, Urdu, Farsi, Hebrew and Yiddish.[2]

In a bidirectional language, the general flow of text proceeds horizontally from right to left, but numbers are written from left to right, the same way as they are written in English. In addition, if an English or another left-to-right language text (addresses, acronyms or quotations) is embedded, it is also written from left to right.

2.2.1 The Arabic Languages and their Writing System

اللغة العربية لغة جميلة

Arabic is a Semitic language that originated with the Arabs of the Hejaz and Nejd regions of Saudi Arabia. There are several spoken dialects of Arabic, but all are derived from the same root: the classical Arabic, which is taught at school in all Arab countries, and is used in all these countries for writing. The written form of the language has different levels of sophistication, depending on the use. These levels range from newspaper style to literary style, passing through technical, business and administration styles.

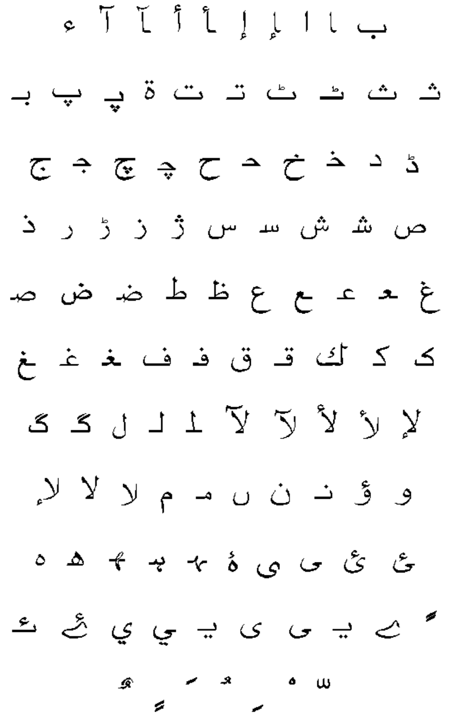


Figure 2-1 Arabic, Farsi and Urdu Characters

2. Arabic is spoken mainly in Algeria, Bahrain, Egypt, Iraq, Israel, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudi Arabia, Sudan, Syria, Tunisia, United Arab Emirates, and Yemen. Urdu is spoken mainly in Pakistan. Farsi is spoken mainly in Iran. Hebrew is spoken mainly in Israel, and Yiddish is spoken mainly in Israel, Europe, and North America.

Arabic Alphabet Characteristics

The basic Arabic alphabet consists of:

- twenty-five consonants
- three long vowels
- one glottal sign
- seven short vowels
- one diacritic mark.

The following characters are added to the basic set, in a way similar to the accented characters in Latin-based languages:

- four carried hamzas
- one diphthong
- two grammatically derived characters
- a connector character, which is included for presentation purposes.

In addition, two special characters (Aleph Wasla and superscripted Aleph) are sometimes used in language education and linguistics.

The Arabic alphabet is also the root from which several other alphabets, such as Farsi and Urdu, are derived. In addition to the characters of the Arabic alphabet, the Farsi and Urdu alphabets have a few more specific characters. Farsi adds four consonants to the Arabic alphabet, and Urdu adds eight.

Character Set Considerations

The Arabic character set has the following characteristics:

- Written Arabic has no equivalent to capital letters.
- The same Arabic characters take on different shapes depending on their position in a text string and on the surrounding characters.
- Arabic script is cursive. Most characters of a word are connected to each other, as in English handwriting.
- Characters can be joined to form ligatures.
- A character can be represented with a vowel or diacritic mark written over or under it.
- Arabic characters have different lengths. As a result, wider characters are sometimes represented on some devices as two coded shapes.

Shapes of the Arabic Characters

Arabic code pages can contain from one to four shapes for each character or ligature, depending on the implementation. The four possible shapes of an Arabic character are:

Isolated	The character is not linked to either the preceding or the following character.
Final	The character is linked to the preceding character but not to the following one.
Initial	The character is linked to the following character but not to the preceding one.

Middle The character is linked to both the preceding and following characters.

Isolated	غ
Final	ـغ
Initial	غـ
Middle	ـغـ

Figure 2-2 The Arabic Character Ghayn in its Different Shapes

In a text string, both the connection capabilities of a character and its neighbours, and its position in the word determine its actual presentation shape.

Numerals Used in Arabic

In countries using Arabic script, the decimal system is in use. In addition to the ‘‘Arabic’’ digits used in the Western world, national digit shapes, known as *Hindi* shapes, are in use. The equivalent of the Arabic digits 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0 are:

Hindi digits (used in Arabic):	١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩ ٠
Farsi digits:	۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۰
Urdu digits:	۱ ۲ ۳ ۴ ۵ ۶ ۷ ۸ ۹ ۰

Figure 2-3 Shape of the National Numbers in Arabic, Farsi and Urdu

2.2.2 The Hebrew Language and its Writing System

ע ב ר י ת

ש פ ה י פ ה

The Hebrew language dates back to biblical times. It remained relatively unchanged for about 2000 years until the end of the nineteenth century, when the birth of the modern Hebrew language took place. Since that time, the Academy of the Hebrew Language has extended traditional Hebrew to include Hebrew words for modern objects and concepts.

Hebrew is used mainly in Israel.

Hebrew Alphabet Characteristics

The Hebrew alphabet uses 27 characters to represent 22 consonants. This is because five consonants have different shapes when they appear at the end of a word.

א ב ג ד ה ו ז

ח ט י ך כ ל ם מ ן נ

ס ע ף פ ץ צ ק ר ש ת

Figure 2-4 Hebrew Alphabet

Vowels are represented in two ways:

- Diacritic marks, called short vowels, may be used above, below or inside characters to represent vowel sounds. Normally, however, the diacritic marks are not shown; the vowel sounds are inferred from the context.
- Some consonants also serve as vowels. When they are used in this way, they are called long vowels.

Vav	ו
Yod	י

Figure 2-5 Hebrew Long Vowels

Written Hebrew has no equivalent to capital letters. Hebrew does not have a cursive script: the letters are not connected. Unlike Arabic, Hebrew letters do not take on different shapes depending on the surrounding letters. The five final shape letters are considered additional, separate letters of the alphabet.

ך	is the final form of Kaf	כ
ם	is the final form of Mem	מ
ן	is the final form of Nun	נ
ף	is the final form of Pe	פ
ץ	is the final form of Tsadi	צ

Figure 2-6 Hebrew Final Letters

2.3 Aspects of Bidirectional Language Writing Systems

This section discusses aspects of bidirectional texts, related to directionality, shaping and national numbers as well as keyboard input and compliance with common user access guidelines. The text attributes described here also pertain to some degree to other complex-text languages such as the languages of Asia (for example, Thai, Lao, Korean).

2.3.1 Bidirectionality

In the context of bidirectionality, the following are key concepts:

- *segments*
- *global orientation*
- *text-types* and associated reordering methods
- *symmetrical swapping.*

These attributes are described below.

Segments

A bidirectional text may consist of a main part that has one directionality (for example, an Arabic text written from right to left), and portions that have an opposite directionality (for example, an English address written from left to right.) The portion of text with a different directionality is called a *segment*. A bidirectional text thus might have a body of right-to-left text with embedded left-to right segments. Sometimes a segment with one directionality might itself have embedded or nested within it an additional segment with an opposite directionality. It is conceptually possible to have many *levels of nesting*; in most cases, however, there are no more than two levels.

One level of nesting is necessary for the entry of numbers within Arabic or Hebrew text. To simulate bidirectional scripts in the following examples, Hebrew and Arabic text is represented by lower-case English letters, while English text is represented by upper-case letters.

In Hebrew, it is customary to write the name of the street before the number of the house, as shown below:

```
b ecnartne 25 teerts elpam
<--------- -> <-----------
```

The street name is entered from right to left. The flow then has to be reversed to allow correct entry of the number from left to right (this being the nested left-to-right segment.) Then the flow must be reversed again to allow the entry of the entrance information from right to left.

Imagine somebody writing a letter in English to somebody who can read Hebrew too, and writing his or her address in Hebrew. In this case, the address in Hebrew is actually a nested segment of the English text.

```
             MY ADDRESS IS b ecnartne 25 teerts elpam THIS MONTH.
              ------------><--------- -> <----------- --------->
NEST LEVEL: 000000000000001111111111221111111111111000000000000
```

Because the nested segment of the address has itself a nested segment (the street number), there are two levels of nesting.

Global Orientation

Bidirectional text may consist of mainly right-to-left text with some left-to-right nested segments (such as an Arabic text with some information in English), or mainly left-to-right with some right-to-left segments (such as an English letter with a Hebrew address nested within it). The predominant direction is called the global orientation; it cannot always be quickly deduced from the general context.

```
FRED DOES NOT BELIEVE taht yas syawla i
```

This sentence has one meaning when the reading is from left to right (Fred does not believe I always say that), and another meaning when read from right to left (I always say that Fred does not believe). In the first half of the above example, the global orientation of the text is left-to-right and in the second half it is right-to-left.

Because the global orientation is not always obvious from the context[3] it must be known to the application developer whose product is processing the bidirectional data.

Note: Not to be confused with the global orientation of the text is the physical orientation of the presentation device. A display terminal has, for example, a right-to-left physical orientation if the first character on the screen is the one in the upper right-hand corner and the general cursor movement is from right to left (and top to bottom.)

Text-types

In a bidirectional text a programmer must clearly distinguish between the *physical order* in which the text is presented, and the *logical order* in which its segments are processed (or pronounced if read aloud). Some segments may need to be reordered to a logical or physical order.

There are different approaches to how bidirectional text is to be reordered, and at present none can be said to be prevalent. The concept *text-type* is used to point to which approach is applicable for a specific text. The physical and logical order and the different text-types are discussed further below.

```
MY WIFE'S NAME IS ilin
```

The global orientation is left-to-right. The first letter in the text is M, followed by Y and so forth. In the *physical order*, after the letters I and S comes the letter i of the segment containing my wife's name in Hebrew. Note, however, that my wife's name is pronounced ''nili''. In the *logical order* the first letter of the name segment is thus the letter n, followed by i, l and i.

Sometimes, for example in on-line help, it is convenient to store the bidirectional text exactly as presented — that is, in the physical order. But if there is an intent to process the text (for example, to sort it), the segments must be stored in their *logical order*. There is no meaning, in the above example, to sort the name ''ilin''. It makes sense to reorder the text, so the directional segment containing the name ''nili'' is inverted, before being stored for further processing. The logical order is the preferred sequence for entering text and for processing. Conceptually, any storage device can be seen as storing the data from left to right. If a programmer wants to perform straightforward processing on the stored text (sorting, collating, indexing) without the need to preprocess each segment, the bidirectional data has to be stored in its logical sequence. This means reversing segments whose direction is opposite to the global orientation.

3. Sometimes it is possible to have a *contextual global orientation*, where the global orientation is set according to the directional characteristic of the first character in the data stream that has a distinct directionality.

Text-types and Reordering Techniques

Different text-types require different approaches to reordering:

visual text-type
: The oldest approach, dating from the time when there was no processing capability at the workstation, is simply to copy the entire screen to storage, and storage to screen (possibly inverting every row, depending on the physical orientation of the screen). It is up to each application programmer to know where the embedded segments are located and to process them accordingly. This text-type is called *visual* because it is a replication of the presented form. Many legacy applications[4] and their files have this type of text.

implicit text-type
: In the *implicit* text-type it is assumed that the letters of the Latin alphabet have a strong inherent left-to-right directionality, and those of the Arabic, Farsi, Urdu and Hebrew alphabet have a strong right-to-left inherent directionality. An algorithm of implicit text processing recognises segments based on their inherent directional characteristics, and segment inversion is performed automatically. The concept of an implicit algorithm is simple to understand. Its main limitation is that it cannot correctly handle some strings that have numbers and intermixed left-to-right and right-to-left letters.

explicit text-type
: The *explicit* text-type assumes that there are additional control characters, embedded in the text, that instruct an explicit algorithm to perform segment inversions, shaping or numeral selections, and other transformations.

Thus, a text with visual text-type is stored in its physical order, and a text with an implicit text-type is stored in its logical order, which is better suited for automatic processing. A text with an explicit text-type is usually stored in logical order, but because of the embedded controls in the text, the automatic processing is not always straightforward.

There is no one type of text that can be used in all cases. The implicit techniques are usually heuristic and thus have some limitations as noted previously. The explicit techniques, while alleviating the limitations of implicit techniques, introduce other limitations such as the need for automatic processes to cope with embedded controls.

One specific technique, the Basic Display Algorithm,[5] tries to be a bridge between the implicit and explicit techniques. In principle it is an implicit reordering algorithm, but it can deal with a few specific directional controls embedded in the text.

There are applications and related databases for all three text-types. It is possible for bidirectional text that is presented one way to be stored in a different layout. A programmer need only know what text-type or reordering algorithm was used, to correctly transform or process the bidirectional text.

4. Legacy applications are those which have been inherited from a prior era. They may be obsolete, but must be supported.
5. The Basic Display Algorithm was initially published in the Directionality Appendix A of the Unicode standard.

Symmetrical Swapping

Some characters, such as the greater-than sign, have an implied directional meaning and have a complementary symmetric character with an opposite directional meaning (the less-than sign). When used within a segment that is presented right-to-left but is inverted (left-to-right) when stored for processing, such a character might have to be replaced by its symmetric sibling to ensure that the correct meaning of the text is preserved. The replacement of such a character by its complement during transformation of a bidirectional text is called *symmetrical swapping*.

Example 2-1 Example of Symmetrical Swapping

On a right-to-left window of the screen, the expression:

```
b < a
```

is read as *a* is *greater* than *b*. In storage the orientation is always left-to-right; the first character in storage is thus *a*, followed by < and then *b*. So the result in storage is:

```
a < b
```

which is of course incorrect. In this case, to preserve the correct meaning of the expression, the < character must be exchanged in storage with >.

Other graphic characters that require symmetrical swapping include the parentheses, square brackets, braces, and so on.

Although symmetrical swapping is a characteristic of bidirectional languages, it is not always mandatory for the software functions that transform different bidirectional-language text layouts. Sometimes this function is performed automatically by the workstation hardware or microcode.

2.3.2 Shaping

Shaping is the process by which characters are rendered in the appropriate presentation forms. This might involve the presentation of characters in a form different from the one in which they are stored. In general, to simplify processing, an unshaped (abstract or basic) representation is used internally. Shaping takes into account the character being shaped and the characters in its vicinity, and replaces its abstract representation (or that of its parts) with the proper shape. Shaping is a characteristic of many complex text languages, in particular the languages of the Middle East.

The Arabic scripts are *cursive*. A writing system is cursive if it is suited to handwriting rather than printing, with adjacent characters in a word connected to each other. Some letters can only connect to the letter on their right. This is the only way in which Arabic script is used, whether in books, newspapers, signs, or workstation displays. (English can be handwritten in a cursive style, for personal communications, but is seldom published or displayed that way. Thus English is not considered a cursive script.)

Shaping in Cursive Script Languages

In cursive scripts, letters might assume different shapes according to their position in the word and to the connectivity properties they and the adjacent letters have. There are as many as four shapes for each letter. As described in **Shapes of the Arabic Characters** on page 8 characters may have initial, middle, final, and isolated forms (not all characters have all forms). Only one shape per letter is represented on Arabic keyboards, but all shapes must be available for presentation. Similarly, in most cases, a cursive language text is not stored with full shapes. Each character has a *base form*, which is an abstraction to allow selection of a cursive character without specifying its shape.

The proper shape can be selected by a *shape determination routine*, which allows for automatic (algorithmic) selection of the appropriate shape according to the context as directed by the software or the user. It may allow for user or software controlled selection of any of the four shapes mentioned above. Alternatively, it may allow transparent throughput of data: that is, it may become temporarily deactivated under software or user control. Whenever cursive-language characters are *folded* by processing to one shape, they must be reshaped using the same algorithm prior to presentation. In some very specific cases, data may be corrupted by this processing, as the algorithm may not be perfectly reversible. As an analogy, in English, converting 12Ab2 to upper case would result in 12AB2; the return to lower case would result in 12ab2, which is not the same as the original.

Though in most cases a cursive language text would be stored in basic shapes only, there are cases where it may be stored with characters shaped as presented, as in the case of messages or on-line help text.

Character Composition, Ligatures and Diacritics

In complex-text languages, it is possible that there is not a one-to-one correspondence between the number of characters of text stored for processing and the number of characters of the presented text. Sometimes two or more characters might be represented by a single glyph occupying one presentation cell:

ligatures

In the cursive languages, ligatures use one glyph to represent two or more specific letters. For example, the ligature *Lamalif* is used to represent the frequently used pair of letters *Lam* and *Alif*.

diacritics

These are marks above, near, within or below a consonant. They are used in bidirectional languages, among other functions, to represent vowels. When kept in storage for processing, these marks occupy physical positions, but if used for representation, they might occupy the same cell as the associated consonants.

As a compromise, given existing limitations (in the graphical capabilities and resolution of the display devices and the number of code points available), bidirectional languages such as Hebrew have in many implementations given up the ability to represent vowels by diacritics. The vowels sounds have to be surmised by readers based on their knowledge of the language and according to the semantics of the text.

However this guesswork is not acceptable for specific applications, such as poetry or processing of a classical text, which requires the use of diacritics. In some complex-text languages, such as Thai, the use of vowel symbols and tone marks is mandatory.

In Arabic, spacing diacritics are currently used as a compromise. In the present Arabic systems, some or all of the Arabic diacritics are implemented as separate characters to be rendered following the character to which the diacritics belong.

2.3.3 National Numbers

In both Latin-based languages and Hebrew, numbers are represented using the so-called Arabic digits (1, 2, 3, 4, 5, 6, 7, 8, 9 and 0). However, the cursive languages, (Arabic, Farsi, and Urdu), as well as many other complex-text languages, have their own national glyphs for digits.[6] The local name for numbers used in the cursive languages is not ‘‘Arabic numbers’’, but *Hindi* or sometimes *Arabic-Indic* numbers. The direction of the numbers is always left-to-right. Mathematical formulae in Arabic are written from right to left and in Farsi they are written from left to right.

It is important to understand that in most cases, the text stored for processing has numbers encoded in their Arabic (western) code. When it comes to presentation, these numbers might be presented using either national glyphs for digits or ordinary Arabic digits, according to the intent of the user or application developer.

2.3.4 Bidirectional Data Entry

To those unfamiliar with bidirectional languages, understanding how segments of text with different directionality can actually be entered from a keyboard is somewhat of a puzzle.

Keying Order

The order in which bidirectional data is typed into a workstation is the order in which the text is meant to be read — the logical order.

Bidirectional Keyboards

The keyboards used for bidirectional languages are similar to those used for English, but on the same keytops on which Latin characters and symbols are engraved, character symbols specific to the other language are added. In the case of the cursive languages, such as Arabic, the character symbols engraved are the basic characters only. Special key combinations are used to switch between the English keyboard layer and the national-language keyboard layer. For example, in some cases, the Hebrew layer is made active, on a Hebrew keyboard, by simultaneously pressing the <Alt> and <Right Shift> keys. Such key combinations are also used to enter appropriate input modes. For example, in some environments, *push mode* is entered by simultaneously pressing <Shift> and <Num Lock>. Push mode is a keyboard input mode in which characters are *pushed* in the direction opposite to the base direction of the segment and the cursor does not move, in the same way the digits behave on the screen of a pocket calculator.

Bidirectional Typing Interfaces

To allow for bidirectional text entry from a keyboard, the interfaces must be able to intercept and process each keystroke. These interfaces can be part of the terminal and associated controller’s hardware or microcode, or they can be a specific routine that is added to the operating system.

There are two typing interfaces to consider:

- manual typing method
- automatic (logical) typing method.

6. Other complex-text languages that have their own national glyphs for decimal digits are Devanagari, Gurmukhi, Gujarati, Oriya, Bengali, Tamil, Telugu, Malayalam, Sinhalese, Khmer (Cambodian), Lao, Mongolian, Chinese, Tibetan and Thai.

Manual Typing Method

In the manual typing method the user informs the system in which direction the characters are to be typed. For mixed-direction typing, the user makes extensive use of the Push and End Push keyboard functions.

The manual method also supports an Automatic Push (Auto Push) mode. When the Auto Push setting is active, the Push Mode is started and terminated automatically, according to the actual characters being typed.

When the manual typing method is active, the keyboard language group and cursor direction are handled separately by the system. This means that the user has separate control for:

- The direction of the field — controlled by the Field Reverse keyboard function.
- The direction of the typed text — controlled by the Push and End Push keyboard functions.
- The keyboard language group — controlled by the keyboard language group switching keys.

Automatic (Logical) Typing Method

This convention provides some automatic handling of directionality. When this method is active, the system determines the directionality of each part of the text (each segment) based on the actual characters being typed, using a set of predefined rules. The method is called logical because the direction of the text is logically deduced based on the language of the characters.

With this method, the system automatically determines how to display characters in the correct order when the user switches keyboard language groups.

Another feature of this method is that it handles text in typing order; that is, the system remembers the order in which the characters were initially typed. It then uses this knowledge along with a set of predefined rules, to determine how the text is displayed, processed and deleted by the application.

If the cursor is in the Home position (the first logical position in the field or window) and a character of a language other than the default language of the current orientation is entered, the screen or window orientation is reversed automatically. That is, if the character entered is Hebrew, the window orientation is right-to-left; if the character is English, the window orientation is left-to-right.

2.3.5 Common User Access and Bidirectional Languages

The basic rule for applications that are to conform to Common User Access guidelines is that ‘‘... All pieces of data must be displayed in the orientation that is correct for the application user. Data input must be supported in the orientation that is natural for users’’.

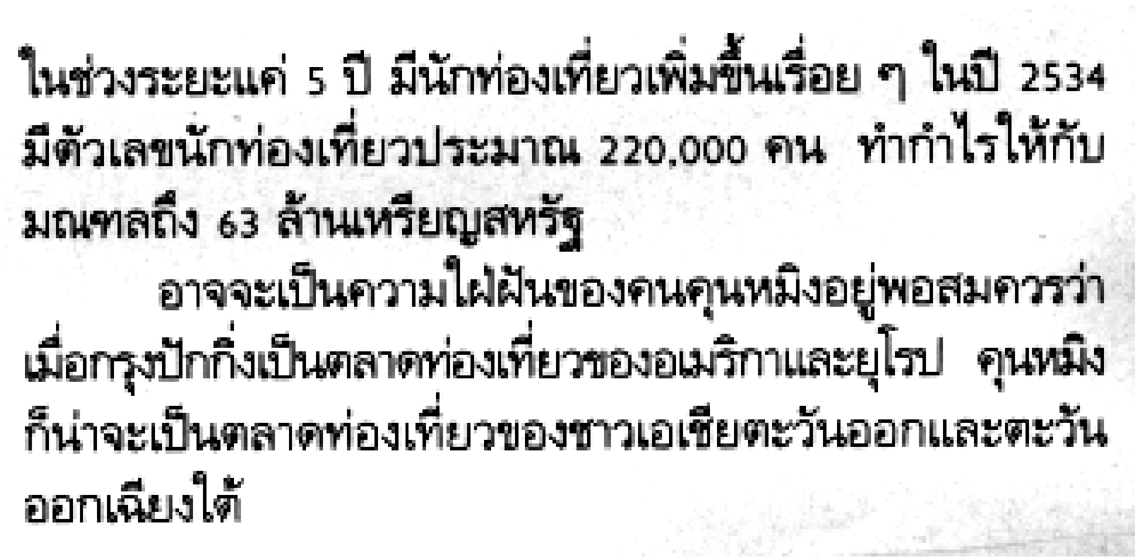
2.4 Thai Language and its Writing System

The Thai language is representative of a class of complex-text languages whose characters are composed of a number of symbols or elements. Thai belongs to the Sino-Tibetan family of languages. Like the Chinese languages, which also belong to this family, Thai is a monosyllabic tone language. While it resembles Chinese structurally and though much of its basic vocabulary is of Chinese origin, it has also been greatly influenced by both Pali and Sanskrit.

The Thai writing system was developed from the Devanagari system, which originated in India and came to Thailand from Cambodia. A major difference between the Chinese and Thai writing systems is that while Chinese makes use of a large number of pictorial symbols, Thai uses an alphabet of consonants, vowels, tone marks, diacritics and special symbols. With some exceptions, a Thai word can be pronounced correctly on sight, in a similar manner to Italian or French.

2.4.1 Writing Thai Characters — Graphic Representation

Thai is written from left to right, without spaces between words. Each word is represented by one or more syllables; each syllable consists of a consonant, a vowel, a tone and a final consonant or a final diacritic. Spaces in the text indicate the ends of phrases or sentences, and are thus used as a form of punctuation. Thus, individual words are recognised only by scanning the text for syllable boundaries. Compared to western writing systems, the composed characters tend to be taller and thinner.



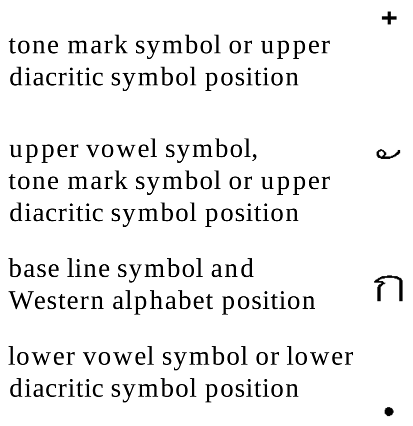
ในช่วงระยะแค่ 5 ปี มีนักท่องเที่ยวเพิ่มขึ้นเรื่อย ๆ ในปี 2534
มีตัวเลขนักท่องเที่ยวประมาณ 220,000 คน ทำกำไรให้กับ
มณฑลถึง 63 ล้านเหรียญสหรัฐ
อาจจะเป็นความใฝ่ฝันของคนคุนหมิงอยู่พอสมควรว่า
เมื่อกรุงปักกิ่งเป็นตลาดท่องเที่ยวของอเมริกาและยุโรป คุนหมิง
ก็น่าจะเป็นตลาดท่องเที่ยวของชาวเอเชียตะวันออกและตะวัน
ออกเฉียงใต้

Figure 2-7 Thai Text Example

A line of Thai text can be considered to be logically divided into four parallel lines:

- the base line, on which consonants, some vowels, some Thai symbols and Thai numbers are written
- the line below the base line, used for writing lower vowels and lower diacritics
- the line above the base line, used for writing upper vowels and upper diacritics

- the line above the upper vowel line, used for writing tone marks and upper diacritics. (If there is no upper vowel, the tone mark or the upper diacritic is written on the upper vowel line.)


tone mark symbol or upper diacritic symbol position
upper vowel symbol, tone mark symbol or upper diacritic symbol position
base line symbol and Western alphabet position
lower vowel symbol or lower diacritic symbol position

2.4.2 Thai Written Symbols

Generally speaking, the more than 2,000 characters in the Thai writing system can be categorised into 20 types of written symbols, with 88 basic symbols:

- 10 base line numerics
- 44 base line consonants
- 3 base line *ancient signs*
- 2 base line special symbols
- 1 base line currency sign
- 1 base line Thai word break character
- 5 base line leading vowels (vowel in front of consonant)
- 3 base line type 1 following vowels
- 1 base line type 2 following vowels
- 2 base line type 3 following vowels
- 1 upper vowel line type 1 upper vowel
- 2 upper vowel line type 2 upper vowel
- 2 upper vowel line type 3 upper vowel
- 1 upper vowel line ancient sign (or upper vowel line type 3 upper diacritic)
- 4 tone mark line tone marks
- 2 tone mark line type 1 upper diacritic symbol
- 1 tone mark line type 2 upper diacritic symbol
- 1 lower vowel line type 1 lower vowel
- 1 lower vowel line type 2 lower vowel
- 1 lower vowel line lower diacritic symbol

Normally, Thai data is encoded using a single-byte code page, where each symbol has an adequate code point. The symbols are used to enter Thai data on a Thai keyboard. Thus the Thai data is stored, for processing purposes, as symbol elements. These elements have to be combined into characters for rendering purposes.

2.4.3 Writing Order

In the most common writing order, first a base line symbol is written, and then optionally, an upper vowel or lower vowel symbol is written above or below it. A tone mark symbol may then optionally be written either above the base line symbol, or above the upper vowel symbol, if present.

This order of writing is taught in Thai elementary schools. However, writing-order inconsistencies exist between individuals. The valid combinations of symbols for Thai composed characters are:

- base line consonant symbol
- base line consonant symbol and tone mark symbol
- base line consonant symbol and upper diacritic symbol
- base line consonant symbol and upper vowel symbol
- base line consonant symbol, upper vowel symbol and tone mark symbol
- base line consonant symbol, upper vowel symbol and upper diacritic symbol
- base line consonant symbol and lower vowel symbol
- base line consonant symbol and lower diacritic symbol
- base line consonant symbol, lower vowel symbol and tone mark symbol.

Any other combinations would be considered invalid.

2.4.4 What is a Thai Character?

From a linguistic or phonetic point of view, the Thai writing system is actually more complex than that described above. Consonants are written on the base line. A middle vowel can be written either before, after or straddling the related consonant. Upper-vowels are written above, and lower vowels below, their related consonant. Vowels are always pronounced and collated after the consonant. The tone mark is usually written after the upper vowel or lower vowel, but some people might write it after the consonant. The left and right pieces of a middle vowel, which straddle a consonant, are included as separate components in some encoding schemes.

To prevent confusion, the term *composed character* is used here for the representation of one syllable at a writing position, and the term *symbol* is used for the components of a composed character.

2.4.5 Thai Numbers

Although Western numerals (Arabic numbers) are now widely used in Thai writing, there are also ten Thai glyphs for numbers. In Thai, the equivalent of the Arabic digits 1, 2, 3, 4, 5, 6, 7, 8, 9 and 0 are respectively:

๑ ๒ ๓ ๔ ๕ ๖ ๗ ๘ ๙ ๐

Figure 2-8 Shape of the National Numbers in Thai

2.4.6 Character Composition

According to the rules for writing Thai, only certain combinations of symbols are possible. When someone fluent in Thai is writing or reading a line, a process of *composition* is taking place. In about 74 percent of cases a character is formed from a single symbol; in about 22 percent of cases, it is formed from two symbols; and in 4 percent of cases it is formed from three symbols.

A Thai speaker does not think of a *composed character* as, for example, an accented character in French. This difference in thinking is reflected in the difference between European and Thai keyboards. In European keyboards, *dead keys* are used to place accents on characters. The dead key is pressed first to show the accent, and then the character key is pressed. The cursor moves only after the character has been entered. All character manipulation is done at the cursor position.

In Thai the consonant or middle vowel is entered first. It is displayed, and the cursor then moves one position to the right. The upper and lower (dead key) vowels and tone marks are then added to the character to the left of the cursor. The rightmost column of positions on the screen is used to display the cursor only, and data is not allowed in this column. Usually vowels and tone marks are stacked on the consonants to compose syllables. The exception is middle vowels, which stay independently at the same level as the consonants.

2.4.7 Thai Character Rendering

Quality font rendering (for example, for desktop publishing), requires additional changes to be made to a Thai composite character form, and sometimes to other characters in its vicinity.

Examples

- Some of the base line symbols that have a descender in the lower position change shape in the presence of a lower vowel.
- Some other base line symbols with a descender do not change their shape. Instead, when these symbols are combined with a lower vowel, the vertical or horizontal position of that lower vowel is changed. Similarly, when some base line symbols with an ascender are combined with an upper vowel, a tone mark or both, the location of the upper vowel, tone mark or both is shifted horizontally.
- The vertical position of a tone mark is dependent upon the presence or absence of an upper vowel. If an upper vowel is not present, the tone mark is positioned at the level that an upper vowel would occupy.
- A specific base line vowel partially overlaps with the associated previous consonant. If the associated consonant does not have an ascender, the vowel is moved up and to the left, to hang over the right side of the previous base line consonant. If the associated consonant has an ascender, the vowel is split into two pieces, with one piece positioned to the left of the ascender and another to the right.

It is thus possible to recognise a similarity between character composition in Thai, and ligatures composition and shaping in bidirectional languages. The character presented is not identical with the symbols stored, so a shaping or composing algorithm must be applied.

Similarly, there are cases where the shaping transformation must not to be performed at rendering, but at a previous stage. When using the high-quality printers adapted for double-byte character set (DBCS), a shaping of characters (maximum three-symbol), is performed as part of the transformation of text to a double-byte encoding scheme. In this case, the text can be considered stored in a *shaped* form for higher-efficiency printing. This resembles the case in which Arabic message text is kept in storage in a shaped layout.

2.5 Korean Language and its Writing System

Korean is the official language of Korea (both south and north) and is spoken by more than 60 million people. The Korean language contains not only Korean words, but also borrows a small number of Chinese-language words. A small number of words are also borrowed from other foreign languages, such as Japanese, English, German, French, and so on.

The official writing system of Korean is known as *Hangul*, which means ‘‘the Korean letter’’ in Korean. It was created and announced by King Sejong and his scholars in 1446. The first standardisation of Hangul was published by the Korean Language Association in 1933.

Hanja, which means ‘‘Chinese letter’’, is a term sometimes used to describe the ideograms used to express words borrowed from Chinese. Likewise, the English alphabet is used to express some English terminology. Nevertheless, contemporary Koreans regard Hangul as the writing system for the Korean language, and as such the Korean writing system is not ideographic.

Korean script uses Arabic figures to represent numbers in most cases. The pronunciation of figures can be expressed using either Hangul or Hanja.

2.5.1 Hangul Writing System

In Hangul, a one-syllable letter may be composed of a cluster of two or three elements, the first of which is always an initial consonant (this can be a *null* consonant, which is not pronounced). The initial consonant is placed at the left, at the upper side or at the upper-left side of the syllable cluster, depending on the second element of the cluster. This second element is always a vowel. The vowels whose representation is based on a vertical line, are positioned to the right of the initial consonant. The vowels whose representation is based on a horizontal line are placed under it. In the case of double vowels with both horizontal and vertical line representation, the initial consonant is placed at the upper-left side and the double vowel is usually placed at the right side and under the consonant. If present, the third element of the cluster is a final consonant, and is always placed under the other two elements of the cluster.

These cluster elements are called *Jamo* in Korean, where *Ja* means consonant and *mo* means vowel.

In standard Hangul, there are 24 basic Jamo elements, of which 14 are used for consonants. In the case of sounds that are not representable by the basic Jamo elements, Hangul grammar allows them to be represented by combining two or more elements. About 27 additional Jamo elements, of which 16 are used for consonants, belong to such combined Jamo elements.

There are 19 permissible initial consonants, 21 vowels and 28 final consonants. The total number of possible combinations of Hangul Consonant-Vowel-Consonant (CVC) is thus 11,172 (19x21x28). Everyday Korean, however, makes do with approximately 2,500 combinations.

Writing Hangul Syllables — Graphic Representation

The major graphic distinction in Hangul is between vowels and consonants. Vowels are based on long horizontal or vertical lines that have distinguishing marks. The *basic vowel* Jamo elements are as follows:

ㅏ ㅑ ㅓ ㅕ ㅗ ㅛ ㅜ ㅠ ㅡ ㅣ

Consonants are represented by more compact, two-dimensional signs; the *basic consonant* Jamo elements are as follows:

ㄱ ㄴ ㄷ ㄹ ㅁ ㅂ ㅅ ㅇ ㅈ ㅊ

ㅋ ㅌ ㅍ ㅎ

The shapes of the consonants were apparently chosen by King Sejong to represent highly stylised pictures of the tongue and mouth when the equivalent sounds are pronounced. For example, the Hangul sign:

ㄴ

which represents a sound equivalent to the English letter *n* has a shape that suggests the tongue-tip raised to touch the front of the palate.

Word Grouping and Direction

In Korean, words are made up of syllables, and the words are separated by spaces. There are two ways in which syllables are juxtaposed to create words and text: vertically (top to bottom and right to left) and horizontally (left to right and top to bottom.)

In South Korea, although newspapers are printed from top to bottom in vertical columns that shift from right to left, horizontal writing is very much in use. The script is Hangul, with occasional Chinese (Hanja) characters as well as Latin-based characters for English text.

In North Korea, newspapers are printed horizontally with Korean script (Hangul only, without Hanja characters).

If, hypothetically, English were written like Hangul, and the vowel O belonged to the set of vowels written under the initial consonant while the vowels A and E were written to the right of the initial consonant, then an English term such as *common market* would be presented in the following manner:

Horizontal direction:

```
C  M     M A  K E
O  O      R    T
M  N
```

Vertical direction of writing:

```
C
O
M

M
O
N


M A
 R

K E
 T
```

Figure 2-9 Korean Writing Direction

The following figure shows actual Hangul Jamo elements and their corresponding composition. The right-most column is an example of vertical writing; the bottom line is an example of horizontal writing.

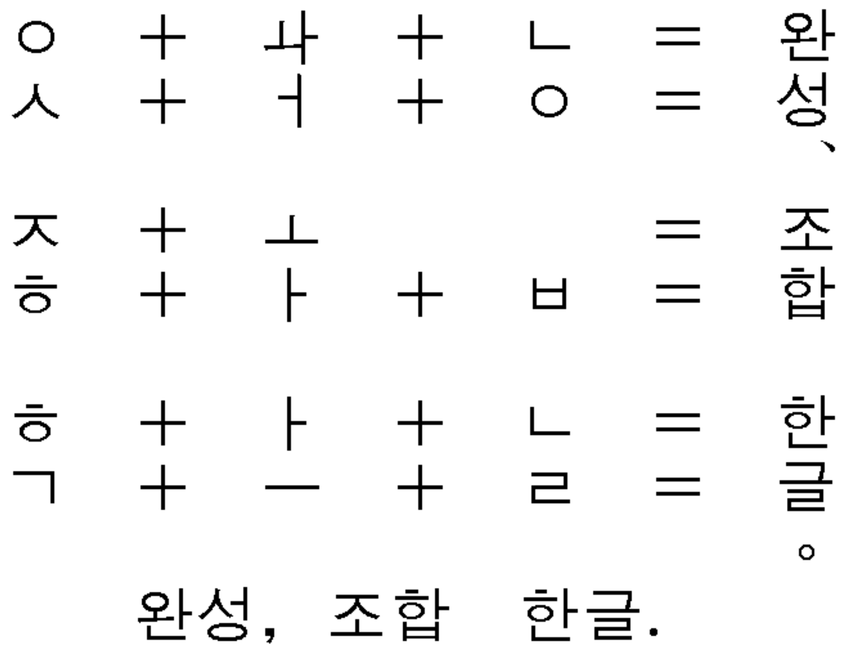


Figure 2-10 Syllable Composition and Writing Direction in Hangul

2.5.2 Character Set Considerations

Korean uses two character sets:

- a single-byte character set (SBCS) made up of Latin characters
- one of two double-byte character set (DBCS):
  - — Completion code:

    a pre-composed Hangul character set of 2,350 syllables
    94 Jamo (including old Jamo) elements
    a Hanja character set
    a non-Hangul and non-Hanja character set containing:
      Latin characters
      Japanese characters (*kanas*)
      other European characters, numerals and symbols.
  - — Combination code: this supports all possible combinations of Hangul CVC by assigning five bits to each CVC. In addition, the code can represent the same Jamo elements, Hanja character set and the non-Hangul and non-Hanja character set of the Completion code.

2.6 Conclusions and Guidelines

Though so different in their appearances, all complex-text languages — the bidirectional ones such as Arabic, Farsi, Urdu, Hebrew and Yiddish, or the languages such as Thai, Lao, or Korean — have a distinct common characteristic: the form of the rendered text is different from that of the stored text. The transformation functions needed to perform the changes between rendered and stored text depend on descriptive information pertaining to the attributes of complex-text languages: global orientation, text-type, symmetrical swapping, shaping and national numbers.

Application developers should be aware of the fact that in the complex-text languages there is a need for transformations between the different text layouts. They should allow for user or system exits to facilitate invoking these transformations, in those places where a transformation might be expected (at input, before output, before a collating process, and so on). Programs must be able to identify the location and content of the complex-text attributes, and be able to change their content if needed.

Just as for any other language, an application meant to be used for complex-text languages should utilise the appropriate language code page and cultural data (date and time layout, collating sequence, monetary layout, and so on).

Application developers should design their products in such a way that they use, as much as possible, the standard functions and controls provided by the operating system services or toolkits for these languages. They might choose to use the APIs offered in the national language versions of the operating system services or toolkits to perform such transformations (when available).

It would be good practice to concentrate all the functions related to National Language in a specific program area for easy maintainability and change support.

Chapter 3

Interface Overview

This chapter outlines the purpose of the *m_*_layout*( ) functions, which facilitate the transformation of the caller's data from one form to the other.

A set of APIs to handle these transformations is defined in Chapter 5. To perform their operations these APIs need descriptive information related to the layouts (layout values of text attributes) and the peculiarities of the characters of the text (the character classifications and mapping).

3.1 Opaque Data

The information needed to perform the transformations is encapsulated in an opaque data type called **LayoutObject**. All locales supported by the *setlocale*( ) function may be associated with a **LayoutObject**. Applications that use the **LayoutObject** functions must first initialise a **LayoutObject**.

Layout values are part of the type **LayoutObject**. A layout value consists of a descriptor and a data type.

The layout values are text attributes and processing indicators needed by the layout transformation functions to relate properly to the text being transformed. The layout values with the data type **LayoutTextDescriptor** have two values: one for input text and one for the transformed text.

3.2 Functions

The *m_create_layout*( ) function initialises a **LayoutObject**. When the function *m_create_layout*( ) is called the locale name is passed to it by an argument of type **AttrObject**. A null argument implies using the locale name of the current locale as set by *setlocale*( ).

The main *m_*_layout*( ) function is *m_transform_layout*( ), which performs layout transformations, such as reordering and shaping, on a string of text encoded in a character encoding scheme. A similar function, *m_wtransform_layout*( ) is provided for text encoded in a wide-character encoding scheme. It also provides information to the application so that it can perform editing, shaping and character composition operations as required.

Other functions associated with the **LayoutObject** type are:

- Free a **LayoutObject** (*m_destroy_layout*( )).
- Set layout values of a **LayoutObject** (*m_setvalues_layout*( )).
- Get layout values of a **LayoutObject** (*m_getvalues_layout*( )).

3.3 Descriptors and Data Types

Table 3-1 lists the standard layout values used by the *m_*_layout*( ) functions associated with a type **LayoutObject**. Each layout value is specified in terms of its descriptor, data type and whether it may be set (S) or got (G) using *m_setvalues_layout*( ) or *m_getvalues_layout*( ) respectively. For some particular national languages or regional groups and for specific implementations, additional layout values, beyond those listed here, may be added.

Defaults have been assigned to the layout values for the C locale, because these locales (in the C library) are the only locales with a consistent behaviour across implementations.

Descriptor	Type	SG
Orientation	**LayoutTextDescriptor**	SG
Context	**LayoutTextDescriptor**	SG
TypeOfText	**LayoutTextDescriptor**	SG
ImplicitAlg	**LayoutTextDescriptor**	SG
Swapping	**LayoutTextDescriptor**	SG
Numerals	**LayoutTextDescriptor**	SG
TextShaping	**LayoutTextDescriptor**	SG
ActiveDirectional	**BooleanValue**	G
ActiveShapeEditing	**BooleanValue**	G
ShapeCharset	**char ***	SG
ShapeCharsetSize	**int**	G
ShapeContextSize	**LayoutEditSize**	G
InOutTextDescrMask	**unsigned long**	SG
InOnlyTextDescr	**unsigned long**	SG
OutOnlyTextDescr	**unsigned long**	SG
CheckMode	**LayoutDesc**	SG
QueryValueSize	**int**	G

Table 3-1 Standard Layout Values

Chapter 4 includes descriptions of the layout values needed by the transformation functions, and the special data structures and types used.

Chapter 4

Header File <sys/layout.h>

This chapter describes the opaque **LayoutObject** type and the other data types and layout values used by the layout services APIs. These are all defined in **<sys/layout.h>**, which is implementation dependent.

4.1 LayoutObject

The **LayoutObject** is an opaque structure that includes values and methods corresponding to a specific locale.

4.1.1 Association with Attribute Objects and Locales

Taking into account emerging trends to facilitate internationalised functions that satisfy multi-locale, multi-threading and multi-node processing, the layout object is associated with a generalised **AttrObject** that might contain other objects beyond the locale object. For information on type **AttrObject**, see the **Distributed Internationalisation Services** snapshot.

In the absence of an **AttrObject**, the locale defaults to the locale supported by the *setlocale*( ) function (see the **XSH**, Issue 4 specification).

4.1.2 LayoutObject Content

LayoutObject contains or points to:

- definitions of the Layout APIs (the *m_*_layout*( ) functions)
- Layout Values:
  — text attributes (LayoutTextDescriptors)
  — processing indicators (for example a value to indicate whether proper rendering of text requires a reordering of directional code elements)
  — different descriptors related to character shaping
- definitions of the specific data type structures needed
- an optional layout category called LO_LTYPE to contain character classifications and mapping with a grammar similar to the locale category LC_CTYPE. The proposal is to define this as a separate layout category included in **LayoutObject**, so that existing applications are not affected. However, LO_LTYPE keywords may be added to existing locales. The description of LO_LTYPE is given in Appendix B.

More detailed descriptions of the different components of **LayoutObject** follow in subsequent sections.

4.2 Layout Values

4.2.1 Descriptors

The different descriptors are described briefly in the following sections. The letters S and G indicate whether the value may be set or retrieved as shown in Table 3-1 on page 28.

For descriptors that do not have an S indicator, the way in which their initial value is set is implementation dependent.

Orientation (SG)

In bidirectional languages, some characters (such as the English letters) are considered to have a *strong* left-to-right orientation; other characters (such as the Arabic characters) are considered strong right-to-left characters; and other characters (such as punctuation marks, spaces, and so on) do not have a strong direction associated with them.

The descriptor Orientation specifies the global directional text orientation. Possible values are:

ORIENTATION_LTR
: Left-to-right horizontal rows that progress from top to bottom.

ORIENTATION_RTL
: Right-to-left horizontal rows that progress from top to bottom.

ORIENTATION_TTBRL
: Top-to-bottom vertical columns that progress from right to left.

ORIENTATION_TTBLR
: Top-to-bottom vertical columns that progress from left to right.

ORIENTATION_CONTEXTUAL
: The global orientation is set according to the direction of the first significant (strong) character.

  If there are no strong characters in the text and the descriptor is set to this value, the global orientation of the text is set according to the value of the descriptor Context. This option is meaningful only for bidirectional text.

The initial state for Orientation is dependent on the **LayoutObject**. If no value is present, the default is ORIENTATION_LTR.

Context (SG)

The descriptor Context is meaningful only if the descriptor Orientation is set to ORIENTATION_CONTEXTUAL. It defines what orientation is assumed when no strong character appears in the text. Possible values are:

CONTEXT_LTR
: In the absence of characters with strong directionality in the text, orientation is assumed to be left-to-right rows progressing from top to bottom.

CONTEXT_RTL
: In the absence of characters with strong directionality in the text, orientation is assumed to be right-to-left rows progressing from top to bottom.

If no value is specified, the default is CONTEXT_LTR.

TypeOfText (SG)

The TypeOfText descriptor specifies the ordering of the directional text. Characters may have a *natural* orientation attached to them as described under **Orientation (SG)** on page 30. An example of how this characteristic could be defined is by the keywords **left_to_right** and **right_to_left** in the layout category LO_LTYPE (see Appendix B). Possible values are:

TEXT_VISUAL
: Code elements are stored in visually ordered segments, which can be rendered without any segment inversion. Practically the whole text could be seen as if there were no sub segments.

TEXT_IMPLICIT
: Code elements are stored in logically ordered segments. Logically ordered means that the order in which the characters are stored is the same as the order in which the characters are pronounced when reading the presented text or the order in which characters would be entered from a keyboard. Logical order (or logical sequence) of characters is necessary for processing purposes, for example, when there is a need to sort or index the data. Segments of reversed orientation are recognised and inverted by a content-sensitive algorithm based on the natural orientation of characters. Because there are several possible algorithms for implicit reordering of directional segments, the ImplicitAlg layout value is used when TypeOfText is set to TEXT_IMPLICIT, to indicate the actual algorithm used.

TEXT_EXPLICIT
: Code elements are stored in logically ordered segments with a set of embedded controls. The explicit algorithm eliminates the ambiguities that might exist in some situations when using an implicit algorithm, but it introduces the need for additional control characters in the data stream. The set of embedded controls for TEXT_EXPLICIT is implementation defined.

Consider the following possible embedded controls:

- Examples of ISO 6429 controls:[7]

  Start Directed String (SDS)
  Start Reversed String (SRS)
  Select Presentation Directions (SPD)
  Graphic Character Combination (GCC)

- Examples of the ISO / IEC 10646 standard (and Unicode) controls: (see the *Basic Display Algorithm* published in Appendix A in the Unicode standard).

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

The **LayoutObject** preserves a bidirectional state across calls to the *m_transform_layout*( ) function. The directional state is reset to the initial state each time TypeOfText is set to any value.

7. This example is based upon the control codes required in the ISO / IEC 6429 standard for handling bidirectional text as published in the ECMA TR / 53 standard. This list is given for illustration only. There is no implied connection between the embedded controls and a specific encoding scheme. The encoding of the above controls depends on the codeset associated with the **LayoutObject**. Any ASCII-based encoding uses the ISO / IEC 6429 standard escape sequence definitions.

Each **LayoutObject** is expected to provide transformation from each of the above types to any of the other types. However, some transformations may cause layout (directional) information to be lost so that text is presented differently after a round trip transformation. This does not imply any data loss, but only possible loss in layout information.

If the TypeOfText value is not specifically stated, the default (for the C locale) is TEXT_IMPLICIT.

ImplicitAlg (SG)

The ImplicitAlg descriptor specifies the type of bidirectional implicit algorithm used in reordering and shaping of directional or context-dependent text.

Possible values of ImplicitAlg are:

ALGOR_IMPLICIT

Directional code elements will be reordered using an implementation-defined implicit directional algorithm when converting to or from an implicit form.

Although the basic algorithm used when ImplicitAlg is set to ALGOR_BASIC, is an implicit algorithm, the fact that it recognises some control characters, allows it to be used even when the TypeOfText descriptor is set to TEXT_EXPLICIT.

Note that when TEXT_EXPLICIT is used in conjunction with ALGOR_BASIC, the controls may temporarily change the values of Swapping, Numerals and TextShaping. The ALGOR_IMPLICIT value may be equal to ALGOR_BASIC for a given implementation. Except in this case, it is not meaningful to have TypeOfText=TEXT_EXPLICIT at the same time as ImplicitAlg=ALGOR_IMPLICIT.

OP ALGOR_BASIC

Directional code elements should be reordered using the basic implicit directional algorithm when converting to and from an implicit form. The *basic reordering algorithm* is the *Basic Display Algorithm* published in the Unicode standard. The basic reordering algorithm is inherently an implicit algorithm, but it may support certain explicit control characters. Among others, the following controls are recognised when reordering with ALGOR_BASIC:

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

All the controls can be found in the referenced Unicode standard.

If the ImplicitAlg value is not specifically stated, the default (for the C locale) is ALGOR_IMPLICIT.

Swapping (SG)

The Swapping descriptor specifies whether symmetric swapping is applied to the text. A list of symmetric swapping characters is given in the ISO / IEC 10646 standard. Possible values are:

SWAPPING_YES

The text conforms to symmetric swapping.

SWAPPING_NO

The text does not conform to symmetric swapping.

If no value is present, the default (for the C locale) is SWAPPING_NO.

Numerals (SG)

The Numerals descriptor specifies the shaping of numerals recognised by the **LayoutObject**. Possible values are:

NUMERALS_NOMINAL
Nominal shaping of numerals using the portable character set (Arabic numerals).

NUMERALS_NATIONAL
National shaping of numerals based on the script of the locale associated with the **LayoutObject** (such as the Thai, Farsi, Hindi, or Bengali script).

An example of how national numbers can be defined is by using the keyword **national_number** in the layout category LO_LTYPE (see Appendix B on page 69).

NUMERALS_CONTEXTUAL
Contextual shaping of numerals depending on the context (script) of surrounding text (such as Hindi numbers in Arabic text and Arabic numbers otherwise).

If no value is specified the default value (for the C locale) is NUMERALS_NOMINAL.

TextShaping (SG)

The descriptor TextShaping specifies the *shaping*; that is, choosing (or composing) the correct shape of the input or output text.

Note: This layout value is important, in particular for languages where the shapes of the characters, when presented, correspond to code points that may be different from the code points of the characters stored for processing:

- In languages such as Arabic or Farsi, the character can have up to four different shapes (see **Shapes of the Arabic Characters** on page 8). In these languages the character is most frequently (but not always) stored and processed using a code point related to a basic shape. Often, but not always, the basic shape chosen is the isolated shape.
- In other complex-text languages, such as Thai and Korean, the shaping is actually a composition process. See Section 2.4.1 on page 18 and Section 2.5.1 on page 23.

Possible values of TextShaping are:

TEXT_SHAPED
The text has presentation form shapes.

TEXT_NOMINAL
The text is in basic form.

TEXT_SHFORM1
The text is in shape form 1.

TEXT_SHFORM2
The text is in shape form 2.

TEXT_SHFORM3
The text is in shape form 3.

TEXT_SHFORM4
The text is in shape form 4.

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

If no value is present, the default value (for the C locale) is TEXT_SHAPED.

In this document, the term *shape form n* is used to mean:

- Arabic Script

shape form 1	initial form
shape form 2	middle form
shape form 3	final form
shape form 4	isolated form

- Thai Script

TEXT_SHAPED	In this form, the characters will be shaped conforming to wtt2.0 specification composed form.
shape form 1	In this form, each character will be shaped into an individual display character occupying a single display cell. Each character has a non-zero width. For example, Mai_Ek is not composed with its preceding character and occupies one display cell.

- Hangul Script

shape form 1	initial consonant-only presentation
shape form 2	vowel-only presentation
shape form 3	final consonant-only presentation
shape form 4	decomposed presentation For example, if each capital letter represents a Jamo, a syllable HAN is presented as three decomposed Jamos of H, A and N.

ActiveDirectional (G)

If the descriptor ActiveDirectional is set (True), then the **LayoutObject** includes knowledge of directional code elements, and proper rendering of text implies reordering of directional code elements. Otherwise the **LayoutObject** does not require any reordering of directional code elements. The way the value of this layout value is set is implementation dependent.

The ActiveDirectional value is guaranteed to remain unchanged for the life of the **LayoutObject**.

ActiveShapeEditing (G)

If the descriptor ActiveShapeEditing is set (True), the **LayoutObject** includes knowledge of context-dependent code elements (an automatic shape determination algorithm) that require shaping for presentation to the ShapeCharset.

The user of a **LayoutObject** is then required to initiate or perform some shaping transformation prior to rendering the text.

Otherwise, the application that uses the **LayoutObject** does not perform shaping, and all code elements may be presented independent of the surrounding characters.

The method used to set ActiveShapeEditing is implementation defined. The ActiveShapeEditing value is guaranteed to remain unchanged for the life of the **LayoutObject**.

ShapeCharset (SG)

The descriptor ShapeCharset specifies the charset of the output text when text is shaped; that is, when ActiveShapeEditing is true. If ActiveShapeEditing is not set (False), in other words, shape editing is a null operation, the ShapeCharset is guaranteed to match the codeset associated with the locale of the **LayoutObject**.

A charset is defined as ‘‘an encoding with a uniform, state-independent mapping from character to code points’’.

A ShapeCharset is a well known name associated with some type of presentation encoding usually used to identify the encoding of a font. Yet, a ShapeCharset need not be a font encoding but may be some intermediate encoding that can then be rendered to a specific font.

Note: **LayoutObject** may be extended to provide an extended layout value, by which the individual glyph metrics may be passed into it.

Since the ShapeCharset is associated with a specific font or glyph encoding, when ActiveShapeEditing is True, the ShapeCharset may (but need not) be the same as the codeset of the locale associated with the **LayoutObject**.

Once chosen, the ShapeCharset is guaranteed to be of a uniform size and state independent, but the size of each ShapeCharset may vary (for example, 8, 16 or another number of bits) so applications should use the ShapeCharsetSize value when doing storage management.

ShapeCharsetSize (G)

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

ShapeContextSize (G)

The ShapeContextSize specifies the size of the context (surrounding code elements) that must be accounted for when performing active shape editing. The ShapeContextSize is defined as structure of type LayoutEditSize, (see discussion on LayoutEditSize in **Type LayoutEditSize** on page 38).

The ShapeContextSize value is guaranteed to remain unchanged for the life of the **LayoutObject**.

InOutTextDescrMask (SG)

This mask is set to tell the layout functions which text descriptors are initialised to valid values when either InOnlyTextDescr or OutOnlyTextDescr are set or queried. For example, if the InOutTextDescrMask is set to denote Orientation and TypeOfText, only these two descriptors are returned when the InOnlyTextDescr is queried. The values used in InOutTextDescrMask are actually a bitwise OR of one or more *classification criteria.*

The way in which these layout values are set is implementation dependent. By default, this descriptor is initialised to indicate that all the text descriptors are to be set and queried.

InOnlyTextDescr (SG)

When this descriptor is set it indicates that the input values of the layout values denoted by the InOutTextDescrMask are set or retrieved when using *m_setvalues_layout*( ) or *m_getvalues_layout*( ) respectively. The way this value is set is implementation dependent.

OutOnlyTextDescr (SG)

When this layout value is set it indicates that the output values of the layout values denoted by the InOutTextDescrMask are set or retrieved when using the *m_setvalues_layout*( ) or *m_getlayoutvalues*( ) respectively.

CheckMode (SG)

The CheckMode layout value indicates the level of checking of the elements in the *InpBuf* for shaping and reordering purposes. It also defines the behaviour of the implicit algorithm with respect to standalone neutral characters (until stabilised by a new strong character).

Possible values of CheckMode are:

MODE_STREAM

The string in the *InpBuf* is expected to have valid combinations of characters or character elements. No validation is needed before shaping and/or combined character cell determination. The only thing validated before the transformation is the current state of the layout object based on previous input data.

The reordering of bidirectional text will assign the nesting level of an unstablised neutral character such that it follows the level of the previous strong character.

When MODE_STREAM is set, it is guaranteed that:

— No [ERANGE] errors will be returned from the *m_*transform_layout*( ) function.

— Each shape associated with a composite sequence will occupy a single display cell.

MODE_EDIT

The shaping of input text may vary depending on locale-specific validation or assumptions.

The reordering of bidirectional text will assign the nesting level of an unstablised neutral character such that it follows the level of the global orientation.

When MODE_EDIT is set:

— [ERANGE] errors may be returned from the *m_*transform_layout*( ) function.

— Not all code elements of a composite sequence may be assumed to occupy a single display cell.

When no value is present, the default of CheckMode (for the C locale) is MODE_STREAM.

QueryValueSize (G)

The user is responsible for his own memory allocation (for the layout values to be queried); therefore he needs to know the actual size of each layout value to be queried.

The name QueryValueSize is defined. This can be ORed with any other name. When *m_getvalues_layout*( ) detects that QueryValueSize is ORed with any name it returns the number of bytes needed to store the value, rather than the value itself. This is to avoid adding a parameter to *m_getvalues_layout*( ).

The following example illustrates the use of QueryValueSize:

```
unsigned long Size;

...

layout[0].name = QueryValueSize | ShapeCharSet;
layout[0].value = &Size;
layout[1].name = 0;
m_getvalues_layout(hlo,layout,&index);
  /*Size should now contain the number of bytes needed
  /*to hold ShapeCharSet*/

...
```

4.2.2 Layout Value Data Types

The following describe the data types used for some of the layout values. All layout values are defined in **<sys/layout.h>**. The content of **<sys/layout.h>** is implementation dependent. In addition the following layout values may be combined (logic OR) into a single type **TextDescriptor**:

Orientation
Context
TypeOfText
ImplicitAlg
Swapping
Numerals
TextShaping

The value of these layout values may also be combined (logic OR) into a single attribute. The layout value AllTextDescriptor can be used to indicate that all **LayoutTextDescriptor** types are set (see **Type LayoutTextDescriptor** on page 38).

Type LayoutValues

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

```
#include <sys/layout.h>

typedef struct{
   LayoutId name;     /* int - the id of the layout value*/
   LayoutValue value; /* void* - Data of layout value item */
   }LayoutValueRec, *LayoutValues;
```

The name element denotes 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 large enough to contain the following: **char***, **long**, **int***, or a pointer to a function. The end of the array is indicated by a name of value zero (0).

The *m_setvalues_layout*( ) function is a convenient way to set the two members of the **LayoutValueRec** structure. This function is usually specified in a stylised manner to minimise the probability of making a mistake. For further information see *m_setvalues_layout*( ) on page 49.

Type LayoutTextDescriptor

The **LayoutTextDescriptor** type is used to identify the attributes of source and target text:

```
#include <sys/layout.h>
typedef int LayoutDesc
typedef struct{
   LayoutDesc inp;    /* Input buffer description */
   LayoutDesc out;    /* Output buffer description */
   } LayoutTextDescriptorRec, *LayoutTextDescriptor;
```

The **inp** and **out** values are 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.

Type LayoutEditSize

The **LayoutEditSize** structure defines the number of surrounding code elements that need to be considered when performing edit shaping:

```
  structure typedef struct{
     int front; /* number of code element in front of the */
                /* edit position in logical order */
     int back;  /* number of code elements following the
                /* edit position in 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 considered when performing edit shaping. The total number of code elements needed to be viewed is:

```
total # of code elements =(front + # code elements in substring + back)
```

If both front and back elements are set to zero, 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.

4.3 Layout Modifiers

Layout modifiers are Layout values in string form.

Each **LayoutObject** consists of several different layout values that are specified in Section 4.2 on page 30 and are initialised at the time the **LayoutObject** is created by the *m_create_layout*( ) function. Yet users may wish to announce an initial layout value that may be different from the default layout value associated with a locale. Thus, the *m_create_layout*( ) function supports a modifier argument that allows the user's default layout values to be passed in a string form. The *m_create_layout*( ) function supports a grammar for the specification of layout values in string form.

The following symbols are used in the proposed grammar for layout modifier strings:

Character	Description
,	Comma
-	Hyphen
/	Solidus (Slash)
;	Semi-colon
=	Equal sign
_	Low line (Underscore)

The following strings are used as prefixes within the grammar definition to mean:

`inout_`	means the value is to be used for both in and out layout values
`in_`	means the value is to be used as an in layout value
`out_`	means the value is to be used as an out layout value.

The proposed grammar is as follows:

```
LSmodifier_string  : '@ls' layout

layout             : layout ',' layout_values
                   | layout_values
                   ;

layout_values      : orientation
                   | context
                   | typeoftext
                   | implicitalg
                   | swapping
                   | numerals
                   | shaping
                   | checkmode
                   | shapcharset
                   ;

orientation        : 'orientation=' inout_orient_value
                   | 'orientation=' in_orient_value ':' out_orient_value
                   ;

inout_orient_value : orient_value
                   ;
```

```
in_orient_value     : orient_value
                    ;

out_orient_value    : orient_value
                    ;

orient_value        : ’ltr’ | ’rtl’ | ’ttblr’ | ’ttbrl’ | ’contextual’
                    ;

context             : ’context=’ inout_context_value
                    | ’context=’ in_context_value ’:’ out_context_value
                    ;

inout_context_value: context_value
                    ;

in_context_value    : context_value
                    ;

out_context_value   : context_value
                    ;

context_value       : ’ltr’ | ’rtl’
                    ;

typeoftext          : ’typeoftext=’ inout_text_value
                    | ’typeoftext=’ in_text_value ’:’ out_text_value
                    ;

inout_text_value    : text_value
                    ;

in_text_value       : text_value
                    ;

out_text_value      : text_value
                    ;

text_value          : ’visual’ | ’implicit’ | ’explicit’
                    ;

implicitalg         : ’implicitalg=’ inout_algor_value
                    | ’implicitalg=’ in_algor_value ’:’ out_algor_value
                    ;

inout_algor_value   : algor_value
                    ;

in_algor_value      : algor_value
                    ;

out_algor_value     : algor_value
```

```
                   ;

algor_value        : ’basic’ | ’implicit’
                   ;

swapping           : ’swapping=’ inout_swap_value
                   | ’swapping=’ in_swap_value ’:’ out_swap_value
                   ;

inout_swap_value   : swap_value
                   ;

in_swap_value      : swap_value
                   ;

out_swap_value     : swap_value
                   ;

swap_value         : ’yes’ | ’no’
                   ;

numerals           : ’numerals=’ inout_num_value
                   | ’numerals=’ in_num_value ’:’ out_num_value
                   ;

inout_num_value    : num_value
                   ;

in_num_value       : num_value
                   ;

out_num_value      : num_value
                   ;

num_value          : ’nominal’ | ’national’ | ’contextual’
                   ;

shaping            : ’shaping=’ inout_shap_value
                   | ’shaping=’ in_shap_value ’:’ out_shap_value
                   ;

inout_shap_value   : shap_value
                   ;

in_shap_value      : shap_value
                   ;

out_shap_value     : shap_value
                   ;

shap_value         : ’shaped’ | ’nominal’ | ’shform1’ | ’shform2’
                   | ’shform3’ | ’shform4’
```

```
                    ;

checkmode           : ’checkmode=’ mode_value
                    ;

mode_value          : ’stream’| ’edit’
                    ;

shapcharset         : ’shapcharset=’ charset_name
                    ;

charset_name        : char_list number
                    | number char_list
                    | char_list
                    | number
                    ;

char_list           : char_list char
                    | char
                    ;

char                : ’A’ | ’B’ | ’C’ | ’D’ | ’E’ | ’F’ | ’G’ | ’H’
                    | ’I’ | ’J’ | ’K’ | ’L’ | ’M’ | ’N’ | ’O’ | ’P’
                    | ’Q’ | ’R’ | ’S’ | ’T’ | ’U’ | ’V’ | ’W’ | ’X’
                    | ’Y’ | ’Z’
                    | ’a’ | ’b’ | ’c’ | ’d’ | ’e’ | ’f’ | ’g’ | ’h’
                    | ’i’ | ’j’ | ’k’ | ’l’ | ’m’ | ’n’ | ’o’ | ’p’
                    | ’q’ | ’r’ | ’s’ | ’t’ | ’u’ | ’v’ | ’w’ | ’x’
                    | ’y’ | ’z’
                    | ’!’ | ’%’ | ’(’ | ’)’ | ’*’ | ’+’ | ’-’ | ’.’
                    | ’_’ | ’?’ |
                    ;

number              : number digit
                    | digit
                    ;

digit               : ’0’ | ’1’ | ’2’ | ’3’ | ’4’ | ’5’ | ’6’ | ’7’
                    | ’8’ | ’9’
                    ;
```

The grammar can be adapted by the implementations to suit their particular needs and their possible extensions to the layout values.

It is expected that higher-level services will use the grammar above to provide users with customisation when running applications.

Examples

1. The Motif text widget may support layout modifiers by means of a resource. If the default for the Hebrew locale is LTR, the following changes it:

   ```
   layoutDirection : right_to_left @ls swapping=yes, numerals=national
   ```

   The key is that the Motif resource value says that right_to_left is the orientation but says nothing about swapping or numerals. The @ls modifier clarifies this.

2. A string could be embedded in a Help repository (such as CDE's help volumes) that describes the layout values to be associated with the text in the help volume. When presented, the help layout values would be passed to the Text widget (see above) in the **layoutDirection** resource. The following is an example of such a string:

   ```
   @ls typeoftext=visual, orientation=rtl, swapping=no,
           numerals=nominal, shapecharset=iso8859-8
   ```

   Note that while this helps speed up presentation of the help text, searches of the help text cannot be made using logical-ordered text (which is the default when entered in an input field). This is because the text (type=visual) has been previously shaped and reordered and thus any text searches need some other processing to account for this.

Chapter 5

Layout APIs

This chapter defines the *m_*_layout*( ) functions in alphabetical order.

NAME

m_create_layout — initialise a layout object

SYNOPSIS

```
#include <sys/layout.h>

LayoutObject m_create_layout(const AttrObject attrobj,
                             const char* modifier);
```

DESCRIPTION

The *m_create_layout*( ) function creates a **LayoutObject** associated with the locale identified by *attrobj*.

The **LayoutObject** is an opaque object containing all the data and methods necessary to perform the layout operations on context-dependent or directional characters of the locale identified by the attrobj. The memory for the **LayoutObject** is allocated by *m_create_layout.*( ) The **LayoutObject** created has default layout values. (If the *modifier* argument is not NULL, the layout values specified by the *modifier* overwrite the default layout values associated with the locale). The defaults are given in Section 4.2 on page 30. Also, internal states maintained by the layout transformation function across transformations, are set to their initial values. The internal state values are implementation dependent and their specifications are not explicitly presented in this document.

The *attrobj* argument is or may be an amalgam of many opaque objects. A locale object is just one example of the type of object that can be attached to an attribute object. The *attrobj* argument specifies a name that is usually associated with a locale category. If *attrobj* is null, the created **LayoutObject** is associated with the current locale as set by the *setlocale*( ) function.

The *modifier* argument can be used to announce a set of layout values when the **LayoutObject** is created. The syntax for this argument is defined in Section 4.3 on page 39.

RETURN VALUE

Upon successful completion, the *m_create_layout*( ) function returns a **LayoutObject** for use in subsequent calls to *m_*_layout*( ) functions. Otherwise the *m_create_layout*( ) function returns (**LayoutObject**)0 and sets *errno* to indicate the error.

ERRORS

The *m_create_layout*( ) function may fail if:

[EINVAL]
The modifier string has a syntax error or it contains unknown layout values.

[EBADF]
The attribute object is invalid or the locale associated with the attribute object is not available.

[EMFILE]
{OPEN_MAX} file descriptors are currently open in the calling process.

[ENOMEM]
Insufficient storage space is available.

NAME

m_destroy_layout — destroy a layout object

SYNOPSIS

```
#include <sys/layout.h>

int m_destroy_layout(const LayoutObject layoutobject);
```

DESCRIPTION

The *m_destroy_layout*( ) function destroys a **LayoutObject** by deallocating the layout object and all the associated resources previously allocated by the *m_create_layout*( ) function.

RETURN VALUE

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

ERRORS

The *m_destroy_layout*( ) function may fail if:

[EBADF]
The attribute object is erroneous.

[EFAULT]
Errors occurred while processing the request.

NAME

m_getvalues_layout — query layout values of a **LayoutObject**

SYNOPSIS

```
#include <sys/layout.h>

int m_getvalues_layout(const LayoutObject layout_object,
                       LayoutValues values, int *index_returned);
```

DESCRIPTION

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

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

The *values* argument specifies the list of layout values that are to be queried. (see **Type LayoutValues** on page 37). Each value element of a **LayoutValueRec** must point to a location where the layout value is stored. That is, if the layout value is of type **T**, the argument must be of type **T***. The values are queried from the **LayoutObject** and represent its current state.

It is the user's responsibility to manage the space allocation for the layout values queried. If the layout value name has QueryValueSize ORed to it, instead of the value of the layout value, only its size is returned. This option can be used by the caller to determine the amount of memory needed to be allocated for the layout values queried (see **QueryValueSize (G)** on page 36).

RETURN VALUE

When the *m_getvalues_layout*( ) function completes without errors a zero is returned. If any value cannot be queried, the index of the value causing the error is returned in *index_returned*, a –1 value is returned and *errno* is set to indicate the error.

ERRORS

The *m_getvalues_layout*( ) function may fail if:

[EINVAL]
: The layout value specified by *index_returned* is unknown or its value is invalid or the argument *layout_object* is invalid. In the case of an invalid *layout_object* argument, the value returned for *index_returned* is –1.

NAME

m_setvalues_layout — set layout values of a **LayoutObject**

SYNOPSIS

```
#include <sys/layout.h>

int m_setvalues_layout(LayoutObject layout_object,
                       const LayoutValues values, int *index_returned);
```

DESCRIPTION

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

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

The *values* argument specifies the list of layout values (see **Type LayoutValues** on page 37). that are to be changed. The values are written into the **LayoutObject** and may affect the behaviour of subsequent layout functions.

Note: Some layout values do alter internal states maintained by a **LayoutObject**.

The *m_setvalues_layout*( ) function can be implemented as a macro that evaluates the first argument twice.

RETURN VALUE

Upon successful completion the requested layout values are set and a value of zero is returned. Otherwise a value of −1 is returned and *errno* is set to indicate the error. If any value cannot be set, none of the layout values is changed and the (zero-based) index of the first value causing the error is returned in *index_returned*.

ERRORS

The *m_setvalues_layout*( ) function may fail if:

[EINVAL]
The layout value specified by *index_returned* is unknown or its value is invalid or the argument layout_object is invalid.

[EMFILE]
{OPEN_MAX} file descriptors are currently open in the calling process.

APPLICATION USAGE

Do not use expressions with side effects such as auto-increment or auto-decrement within the first argument to the *m_setvalues_layout*( ) function.

NAME

m_transform_layout — layout transformation

SYNOPSIS

```
#include <sys/layout.h>

int m_transform_layout(
        LayoutObject   layout_object,
        const char    *InpBuf,
        const size_t   InpSize,
        void          *OutBuf,
        size_t        *Outsize,
        size_t        *InpToOut,
        size_t        *OutToInp,
        unsigned char *Property,
        size_t        *InpBufIndex
        );
```

DESCRIPTION

This function performs layout transformations (reordering, shaping, cell determination) or it may provide additional information needed for layout transformation (such as the expected size of the transformed layout, the nesting level of different segments in the text and cross references between the locations of the corresponding elements before and after the layout transformation). Both the input text and output text are character strings.

The *m_transform_layout*( ) function transforms the input text in *InpBuf* according to the current layout values in *layout_object*. Any layout value whose value type is **LayoutTextDescriptor** describes the attributes of the *InpBuf* and *OutBuf* arguments. If the attributes are the same for both *InpBuf* and *OutBuf*, a null transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be NULL, unless there is a need to reset the internal state.

The *InpSize* argument is the number of bytes within *InpBuf* to be processed by the transformation. Its value will not change after return from the transformation. *InpSize* set to –1 indicates that the text in *InpBuf* is delimited by a NULL code element. If *InpSize* is not set to –1, it is possible to have some NULL elements in the input buffer. This might be used, for example, for a ‘‘one shot’’ transformation of several strings, separated by NULLs.

Outputs of this function may be one or more of the following depending on the setting of the arguments:

OutBuf
: Any transformed data is stored in *OutBuf*, converted to ShapeCharset.

Outsize
: The number of bytes in *OutBuf*.

InpToOut
: A cross reference from each *InpBuf* code element to the transformed data. The cross reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

OutToInp
: A cross reference to each *InpBuf* code element from the transformed data. The cross reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points

to (and not necessarily starting from the beginning of the *InpBuf*).

Property

A weighted value that represents peculiar input string transformation properties with different connotations as explained below.

If this argument is not a NULL pointer, it represents an array of values with the same number of elements as the source substring text before the transformation. Each byte will contain relevant ''property'' information of the corresponding element in *InpBuf starting from the element pointed by InpBufIndex*. The four rightmost bits of each ''property'' byte will contain information for bidirectional environments (when ActiveDirectional is True) and they will mean ''NestingLevels''. The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf starting from the element pointed by InpBufIndex*. If ActiveDirectional is false the content of NestingLevel bits will be ignored. The leftmost bit of each ''property'' byte will contain a ''new cell indicator'' for composed character environments, and will have a value of either 1 (for an element in *InpBuf* that is transformed to the beginning of a new cell) or zero (for the ''zero-length'' composing character elements, when these are grouped into the same presentation cell with a non-composing character). Here again, each element of ''property'' pertains to the elements in the *InpBuf starting from the element pointed by InpBufIndex*. (Remember that this is not necessarily the beginning of *InpBuf*). If none of the transformation properties is required, the argument *Property* can be NULL.

The use of ''property'' can be enhanced in the future to pertain to other possible usage in other environments.

The *InpBufIndex* argument is an offset value to the location of the transformed text. When *m_transform_layout*( ) is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*). At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

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.

The layout object maintains a *directional state* that keeps track of directional changes, based on the last segment transformed. The directional state is maintained across calls to the layout transformation functions and allows stream data to be processed with the layout functions. The directional state is reset to its initial state whenever any of the layout values TypeOfText, Orientation or ImplicitAlg is modified by means of a call to *m_setvalues_layout.*( )

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

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 ShapeCharset 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 codeset 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 bytes. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set (True) the *OutBuf* should be allocated to contain at least the *InpSize* multiplied by 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 calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged. If *OutSize* = NULL, the [EINVAL] error condition should be returned.

If the *InpToOut* argument is not a NULL pointer, it points to an array of values with the same number of bytes in *InpBuf* starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer. On output, the *n*th value in *InpToOut* corresponds to the *n*th byte in *InpBuf*. This value is the index (in units of bytes) in *OutBuf* that identifies the transformed ShapeCharset element of the *n*th byte in *InpBuf*. In the case of multibyte encoding, the index points (for each of the bytes of a code element in the *InpBuf*) to the first byte of the transformed code element in the *OutBuf*.

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 points to an array of values with the same number of bytes as contained in *OutBuf*. On output, the *n*th value in *OutToInp* corresponds to the *n*th byte in *OutBuf*. This value is the index in *InpBuf*, starting with the byte pointed to by *InpBufIndex,* that identifies the logical code element of the *n*th byte in *OutBuf*. In the case of multibyte encoding, the index will point for each of the bytes of a transformed code element in the *OutBuf* to the first byte of the code element in the *InpBuf*.

OutToInp may be specified as NULL if no index array from *OutBuf* to *InpBuf* is desired.

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

RETURN VALUE

If successful, the *m_transform_layout*( ) function returns zero. If unsuccessful, the returned value is −1 and the *errno* is set to indicate the source of error. 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 [E2BIG] is returned in *errno*.

ERRORS

The *m_transform_layout*( ) function may fail if:

[EILSEQ]
Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InpBufIndex* argument is set to indicate the code element causing the error. The suspect code element is either a valid code element but cannot be shaped into the ShapeCharset layout value, or is an invalid code element not defined by the codeset of the locale of *layout_object*. The *mbtowc*( ) and *wctomb*( ) functions, when used in the same locale as the **LayoutObject**, can be used to determine if the code element is valid.

[E2BIG]
The output buffer is full and the source text is not entirely processed.

[EINVAL]
Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains NULL.

[ERANGE]
More than 15 embedding levels are in source text or *InpBuf* contain unbalanced directional layout information (push/pop) or an incomplete composite sequence has been detected in the input buffer at the beginning of the string pointed to by *InpBufIndex*.

Note: An incomplete composite sequence at the end of the input buffer is not always detectable. Sometimes, the fact that the sequence is incomplete will only be detected when additional character elements belonging to the composite sequence are found at the beginning of the next input buffer.

[EBADF]
The layout values are set to a meaningless combination or the layout object is not valid.

APPLICATION USAGE

A **LayoutObject** will have a meaningful combination of default layout values. Whoever chooses to change the default layout values is responsible for making sure that the combination of layout values is meaningful. Otherwise, the result of *m_transform_layout*( ) might be unpredictable or implementation-specific with *errno* set to [EBADF].

NAME

m_wtransform_layout — layout transformation for wide character strings

SYNOPSIS

```
#include <sys/layout.h>

int m_wtransform_layout (
        LayoutObject     layout_object,
        const wchar_t   *InpBuf,
        const size_t     InpSize,
        void            *OutBuf,
        size_t          *Outsize,
        size_t          *InpToOut,
        size_t          *OutToInp,
        unsigned char   *Property,
        size_t          *InpBufIndex
        );
```

DESCRIPTION

This function performs layout transformations (reordering and shaping, cell determination) or it may provide additional information needed for layout transformation (such as the expected size of the transformed layout, the nesting level of different segments in the text and cross references between the locations of the corresponding elements before and after the layout transformation). Both the input text and output text are wide character strings.

The *m_wtransform_layout*( ) function transforms the input text in *InpBuf* according to the current layout values in *layout_object*. 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*, a null transformation is performed with respect to that specific layout value.

The *InpBuf* argument specifies the source text to be processed. The *InpBuf* may not be NULL, unless there is a need to reset the internal state.

The *InpSize* is the number of characters within *InpBuf* to be processed by the transformation. Its value will not change after return from the transformation. *InpSize* set to –1 indicates that the text in *InpBuf* is delimited by a NULL code element. If *InpSize* is not set to –1, it is possible to have some NULL elements in the input buffer. This might be used, for example, for a ‘‘one shot’’ transformation of several strings, separated by NULLs.

Outputs of this function may be one or more of the following depending on the setting of the arguments:

OutBuf
: Any transformed data is stored in *OutBuf*, converted to ShapeCharset.

Outsize
: The number of wide characters in *OutBuf*.

InpToOut
: A cross reference from each *InpBuf* code element to the transformed data. The cross reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

OutToInp
: A cross reference to each *InpBuf* code element from the transformed data. The cross reference relates to the data in *InpBuf* starting with the first element that *InpBufIndex* points to (and not necessarily starting from the beginning of the *InpBuf*).

Property

A weighted value that represents peculiar input string transformation properties with different connotations as explained below.

If this argument is not a NULL pointer, it represents an array of values with the same number of elements as the source substring text before the transformation. Each byte will contain relevant ''property'' information of the corresponding element in *InpBuf starting from the element pointed by InpBufIndex*. The four rightmost bits of each ''property'' byte will contain information for bidirectional environments (when ActiveDirectional is True) and they will mean ''NestingLevels''. The possible value from 0 to 15 represents the nesting level of the corresponding element in the *InpBuf starting from the element pointed by InpBufIndex*. If ActiveDirectional is false the content of NestingLevel bits will be ignored. The leftmost bit of each ''property'' byte will contain a ''new cell indicator'' for composed character environments, and will have a value of either 1 (for an element in *InpBuf* that is transformed to the beginning of a new cell) or zero (for the ''zero-length'' composing character elements, when these are grouped into the same presentation cell with a non-composing character). Here again, each element of ''property'' pertains to the elements in the *InpBuf starting from the element pointed by InpBufIndex*. (Remember that this is not necessarily the beginning of *InpBuf*). If none of the transformation properties is required, the argument *Property* can be NULL.

The use of ''property'' can be enhanced in the future to pertain to other possible usage in other environments.

The *InpBufIndex* argument is an offset value to the location of the transformed text. When *m_wtransform_layout*( ) is called, *InpBufIndex* contains the offset to the element in *InpBuf* that will be transformed first. (Note that this is not necessarily the first element in *InpBuf*.) At the return from the transformation, *InpBufIndex* contains the offset to the first element in the *InpBuf* that has not been transformed. If the entire substring has been transformed successfully, *InpBufIndex* will be incremented by the amount defined by *InpSize*.

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 outputs above the *layout_object* maintains a directional state across calls to the transform functions. The directional state is reset to its initial state whenever any of the layout values TypeOfText, Orientation or ImplicitAlg is modified by means of a call to *m_setvalues_layout*( ).

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

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 ShapeCharset 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 codeset of the locale associated with the attribute object indicated during the creation of the Layout Object.

On input, the *OutSize* argument specifies the size of the output buffer in number of wide characters. The output buffer should be large enough to contain the transformed result; otherwise, only a partial transformation is performed. If the ActiveShapeEditing layout value is set (True) the *OutBuf* should be allocated to contain at least the InpSize multiplied by ShapeCharsetSize.

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

When the *OutSize* argument is specified as zero, the function calculates the size of an output buffer large enough to contain the transformed text, and the result is returned in this field. The content of the buffers specified by *InpBuf* and *OutBuf*, and the value of *InpBufIndex*, remain unchanged. If *OutSize* = NULL, the [EINVAL] error condition should be returned.

If the *InpToOut* argument is not a NULL pointer, it points to an array of values with the same number of wide characters in *InpBuf* starting with the one pointed by *InpBufIndex* and up to the end of the substring in the buffer. On output, the *n*th value in *InpToOut* corresponds to the *n*th wide character in *InpBuf*. This value is the index (in units of wide characters) in *OutBuf* that identifies the transformed ShapeCharset element of the *n*th wide character 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 points to an array of values with the same number of wide characters as contained in *OutBuf*. On output the *nth* value in *OutToInp* corresponds to the *nth* wide character in *OutBuf*. This value is the index in *InpBuf*, starting with the wide character pointed to by *InpBufIndex*, that identifies the logical code element of the *n*th wide character in *OutBuf*.

OutToInp may be specified as NULL if no index array from OutBuf to InpBuf is desired.

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

RETURN VALUE

If successful, the *m_wtransform_layout*( ) function returns a zero. If unsuccessful, the returned value is −1 and the *errno* is set to indicate the source of error. 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 [E2BIG] is returned in *errno*.

ERRORS

The *m_wtransform_layout*( ) function may fail if:

[EILSEQ]
Transformation stopped due to an input code element that cannot be shaped or is invalid. The *InfBufIndex* argument is set to indicate the code element causing the error. The suspect code element is either a valid code element but cannot be shaped into the ShapeCharset layout value or is an invalid code element not defined by codeset of the locale of *layout_object*. The *mbtowc*( ) and *wctomb*( ) functions, when used in the same locale as the *layoutobject*, can be used to determine if the code element is valid.

[E2BIG]
The output buffer is full and the source text is not entirely processed.

[EINVAL]
Transformation stopped due to an incomplete composite sequence at the end of the input buffer, or *OutSize* contains NULL.

[ERANGE]
More than 15 embedding levels are in source text or *InpBuf* contain unbalanced directional layout information (push/pop) or an incomplete composite sequence has been detected in the input buffer at the beginning of the string pointed to by *InpBufIndex*.

Note: An incomplete composite sequence at the end of the input buffer is not always detectable. Sometimes, the fact that the sequence is incomplete will only be detected when additional character elements belonging to the composite sequence are found at the beginning of the next input buffer.

[EBADF]
The layout values are set to a meaningless combination or the layout object is not valid.

APPLICATION USAGE

A **LayoutObject** will have a meaningful combination of default layout values. Whoever chooses to change the default layout values is responsible for making sure that the combination of layout values is meaningful. Otherwise, the result of *m_transform_layout*( ) might be unpredictable or implementation-specific with *errno* set to [EBADF].

EXAMPLES

The following example illustrates what the different arguments of *m_wtransform_layout*( ) look like when a string in *InpBuf* is shaped and reordered into *OutBuf*. Upper-case letters in the example represent left-to-right letters while lower-case letters represent right-to-left letters.
xyz represents the shapes of cde.

```
Position:           0123456789
InpBuf:             AB cde 12Z

Position:           0123456789
OutBuf:             AB 12 zyxZ

Position:           0123456789
InpToOut:           0128765349

Position:           0123456789
OutToInp:           0127865439

Position:           0123456789
Property.NestLevel: 0001111220
Property.CelBdry:   1111111111
```

The values (encoded in binary) returned in the *Property* argument define the directionality of each code element in the source text as defined by the type of algorithm used within the *layout_object*. While the algorithm may be implementation dependent the resulting values and levels are defined such as to 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.
- The Orientation layout value setting determines the initial level (0 or 1) used.

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

Example of Possible Presentation Algorithm

The following is an example of a standard presentation algorithm that handles nesting correctly. The goal of the algorithm is ultimately to return to a zero nest level.

Note: More efficient algorithms do exist; the following is provided for clarity rather than for efficiency.

1. Search for the highest nest level in the string.
2. Reverse all surrounding code elements of the same level. Reduce the nest 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 of the example from above:

```
Position:           0123456789    0123456789  0123456789
InpBuf:             AB cde 12Z    AB cde 21Z  AB 12 edcZ
Property.NestLevel: 0001111220    0001111110  0000000000
Property.CellBdry:  1111111111    1111111111  1111111111
```

5.1 Notes

1. Using the *InpBufIndex* content and analysing the content of *errno*, in the case that an exceptional transformation event has occurred, the application can take steps to correct the reason for the exceptional events, such as enlarging the size of the *OutBuf* if [E2BIG] was returned, and it has the latitude to choose between different possible ulterior actions:

   — reset the value of *InpBufIndex* to the value it had before the transformation was called, and restart the whole transformation again with the original contents of *InpBuf* either unaltered or refreshed

   or:

   — leave the value of *InpBufIndex* unaltered and let the *m_transform_layout*( ) function perform the transformation again from the point it has been interrupted.

   Naturally the content of *OutBuf* will vary according to the option chosen and the caller to the *m_transform_layout*( ) function will be aware of it and use it as suitable.

2. The ‘‘front’’ component of the layout value ShapeContextSize will be used for Thai in order to determine the maximum number of character elements needed before the first character element pointed to by *InpBufIndex* in order to correctly determine the cell group for the character elements at the beginning of the transformed substring.

3. In Thai there is a possibility that though a sequence of character elements is correctly composing a Thai composite element, there is an additional composing character immediately following them that belongs to the same cell. Because this condition can be detected when processing the transformation of the subsequent substring where the ‘‘additional’’ composing element will be the first one, the [EINVAL] error value will *not* be set at the end of processing Thai substrings that end with a valid combination (that can potentially be incomplete after all).

4. An example of possible use of MODE_EDIT for shaping based on locale specific assumptions is for scripts that have various shapes — initial, middle, isolated, final, and so on — and the shape of a character depends on its connecting ability to its neighbouring characters, if present. Assume that *InpBuf* contains newly keyed in data and the *m_*transform_layout*( ) function is used to shape this data in order to present it on the screen. If the value of CheckMode is MODE_EDIT, the last character entered (to which *InpBufIndex* points) will be shaped as if it is followed by another connect-to-the-right letter (as it happens in most of the cases) causing the transformation to use initial or middle shapes. If CheckMode were not set to MODE_EDIT such text would be transformed into isolated or final shapes (which may need to be reshaped to initial or middle shape once the next connect-to-the right letter belonging to the same word will be keyed in).

   The MODE_EDIT checking is also useful for scripts where determination of display cells is needed. Specifically where composite sequences may span display cell boundary.

5. The SaraAm character from Thai can be shaped such that it affects the preceding character (that is, in some cases, the SaraAm is split at output; part of it is composed with the preceding consonant, and part of it is composed as a separate cell). Below is an example that shows how the transform function would behave in the case of this character.

Let assume ’a’ is a consonant and ’b’ is a SaraAm character in the input buffer:

```
Inp Buf:  | a | b | c |   | d |
```

And, actual screen will look as follows:

```
| b1|   | c2|   | d2|
| a | b2| c1|   | d1|  <-- baseline symbol/consonant
|   |   |   |   | d3|
```

The results of a transform would be expected to behave as follows:

```
Position:              | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 |
Inp Buf:               | a | b | c |   | d |
Inp BufIndex:            0
Out Buf:               | a | b1| b2| c1| c2|   | d1| d2| d3|
InpToOut:              | 1 | 3 | 4 | 6 | 7 |
OutToInp:              | 1 | 2 | 2 | 3 | 3 | 4 | 5 | 5 | 5 |
Property.CellBdry:     | 1 | 1 | 1 | 1 | 1 |
Property.NestLevel:    | 0 | 0 | 0 | 0 | 0 |
InputBufIndex return:  10
```

Note that the Property display cell boundary (Property.CelBdry) information says that the SaraAm character is in its own display cell. Yet it’s shape actually affects the preceding display cell. But also note that the *InpToOut* index for the SaraAm (’b’) character points to the 3rd position in the output buffer, while the indication that "part" of "b" in the form of "b1" appears in the same cell as "a" is reflected in the second position of *OutToInp*. Thus determination of the display cell boundary within the output buffer requires analysis of both the Property display cell information (that relates to the input buffer), as well as the *OutToInp* and the *InpToOut* buffer.

Implementation Example

OP This appendix contains examples of possible uses of the layout transformation APIs in Motif 2.0. Example A-1 is an internationalised program ("hello world") that has been modified in such a way that using modifiers it can be localised in a BIDI environment at run time. Example A-2 on page 65 and Example A-3 on page 66 show possible usages of the layout APIs to perform the transformations that are depicted in Figure 1-1 on page 2. Example A-2 handles conversion from storage (implicit in this case) to type **XmString** (ISO / IEC 6429 standard controls). Example A-3 handles conversions from type **XmString** to the display (visual) buffer. In Example A-2 and Example A-3 four controls are used:

<RTL>	Right-to-left explicit control.
<LTR>	Left-to-right explicit control.
<push>	Start reverse global direction.
<pop>	End push.

Example A-1 Internationalised Motif Program to Support Layout APIs

```
/*
 *  The following is an example of an internationalised Motif hello
 *  world program that has been modified to support Bidirectional text.
 *  The layout values are passed via a modifier string thus allowing
 *  the internationalised program to be localised at run time.
 */

#include <stdio.h>

#include <Xm/Xm.h>
#include <Xm/Form.h>
#include <Xm/PushB.h>
#include <locale.h>
#include <sys/layout.h>

static void helloworld_button_activate();

Display             *display;
XtAppContext        app_context;
LayoutObject        plh;
char                *inp;

/*
 *  Define layoutModifier resource to allow localisation
 *  of the text being passed to hello world.
 */
#define  XtNlayoutModifier   "layoutModifier"
#define  XtCLayoutModifier   "LayoutModifier"
char                *lo_modifier;
static XtResource lo_resources[] = {
  { XtNlayoutModifier, XtCLayoutModifier, XmRString, sizeof(String),
        0, XmRString, (String) NULL}
};
```

```
/*
 *  Main program
 */
int main(argc, argv)
unsigned int argc;
char **argv;
{
    /*
     *   Declare the variables to contain the two widget ids.
     */
    Widget toplevel, bulletin, pushb;
    Arg  arglist[3] ;
    int  n;
```

```
    if (argc > 1)
    inp = argv[1];
    else {
    fprintf(stderr,"Incorrect number of arguments0);
    exit(0);
    }
```

```
    XtSetLanguageProc(NULL,NULL,NULL);
```

```
    XtToolkitInitialize();
```

```
    app_context    = XtCreateApplicationContext();
```

```
    display = XtOpenDisplay(app_context, NULL, argv[0],
                            "helloworldclass",
                             NULL, 0, &argc, argv);
    if (display    == NULL) {
    fprintf(stderr,     "%s:  Can’t open display0, argv[0]);
    exit(1);
    }
```

```
    n =  0;
    XtSetArg(arglist[n], XmNallowShellResize, True);  n++;
    toplevel = XtAppCreateShell(argv[0], NULL,
                                applicationShellWidgetClass,
                                display, arglist,  n);
```

```
    /* Set up the lo_modifier */
    XtGetApplicationResources(toplevel,     &lo_modifier, lo_resources,
                   XtNumber(lo_resources), NULL, 0);
```

```
    XtSetArg (arglist[0], XmNwidth, 200);
    XtSetArg (arglist[1], XmNheight, 100);
    bulletin = XmCreateBulletinBoard(toplevel, "bulletin", arglist, 2);
    XtManageChild(bulletin);

    XtSetArg (arglist[0], XmNlabelString,
              XmStringCreateLtoR("Press Button...",
         XmSTRING_DEFAULT_CHARSET));
    pushb = XmCreatePushButtonGadget(bulletin, "push", arglist, 1);
    XtAddCallback (pushb, XmNactivateCallback,
                   helloworld_button_activate,NULL);
    XtManageChild(pushb);

    /*
     *  Realise the top-level widget. This will
     *  cause the entire "managed"
     *  widget hierarchy to be displayed.
     */

    XtRealizeWidget(toplevel);

    /*
     *  Loop and process events.
     */

    XtAppMainLoop(app_context);

    /* UNREACHABLE */
    return (0);
}

BooleanValue reorder_or_shaping( plh )
LayoutObject plh;
{
    size_t index = 0;
    LayoutValueRec lo_values[3];
    int rc = 0;
    BooleanValue directional;
    BooleanValue shaping;

    lo_values[0].name = ActiveShapeEditing ;
    lo_values[0].value = (LayoutValue) &shaping ;
    lo_values[1].name = ActiveDirectional ;
    lo_values[1].value = (LayoutValue) &directional ;
    lo_values[2].name = 0;

    rc = m_getvalues_layout(plh, lo_values, &index);
    if (rc)
    {
      perror("m_getvalues_layout");
      exit(1);
    }

    return ( shaping || directional );
}
```

```
static void helloworld_button_activate(     widget,   tag,
                                        callback_data )
    Widget    widget;
    char *tag;
    XmAnyCallbackStruct *callback_data;
{
    Arg  arglist[2];

    static int call_count = 0;
    char *label;
    size_t inpsize, label_size;
    size_t index = 0;
    int rc = 0;
    int mask = 0;

    call_count += 1 ;
    switch ( call_count )
    {
    case 1:
        plh   = m_create_layout(NULL,  lo_modifier);

        if ( ! reorder_or_shaping(plh) )
             {
             XtSetArg( arglist[0], XmNlabelString,
              XmStringCreateLocalized(inp));
             }
        else
        {
            inpsize = strlen(inp);
             label = (char *)malloc(inpsize+1);
             memset(label, 0, inpsize+1);
             rc = m_transform_layout(plh, inp, inpsize,
              label, &label_size, NULL, NULL,    NULL, &index);
             if (rc)
             {
              perror("m_transform_layout");
              exit(1);
             }

             XtSetArg( arglist[0], XmNlabelString,
              XmStringCreateLocalized(label));
             free(label);
        }
        m_destroy_layout(plh);
        XtSetArg( arglist[1], XmNx,    11 );
        XtSetValues( widget, arglist, 2 );
        break ;
    case 2:
        exit(0);
        break ;
    }
}
```

Example A-2 Storage to Explicit Controls

```
#include <sys/layout.h>
#include <errno.h>
#include <locale.h>

main
{
  LayoutObject plh;
  int rc=0;
  LayoutValues layout;
  LayoutTextDescriptor descr;
  int index;
  char *inpbuff="AB cde 123Z";
  char *outbuff;
  size_t inpsize;
  size_t outsize;
  size_t index=0;

  /* Create layout object */
  plh = m_create_layout(NULL, NULL);

  /* Set the internal "in" and "out" values of the TypeOfText */
  /* descriptor to initiate the "implicit" to "explicit" */
  /* transformation, with character shaping and national numerals */

  layout=malloc(2*sizeof(LayoutValueRec));
  descr=malloc(sizeof(LayoutTextDescriptorRec));

  layout[0].name=TypeOfText;
  layout[0].value=(caddr_t)descr;
  layout[1].name=0;

  descr->in=TEXT_IMPLICIT;
  descr->out=TEXT_EXPLICIT;

  rc = m_setvalues_layout(plh,layout,&index);
  if (rc)
  {
      perror("m_setvalues_layout : ");
      exit(0);
  }

  /* Set the input buffer size */

  inpsize = strlen(inpbuff);

  /* Now, allocate an outbuff large enough to contain the original
     buffer plus the embedded escapes */

  outbuff = (char *)malloc(inpsize*sizeof(char)*2);
  rc = m_transform_layout(plh, inpbuff, inpsize, outbuff, &outsize,
                          NULL, NULL, NULL, &index);

  if (rc)
  {
      perror("m_transform_layout : ");
      exit(0);
  }

  free(layout);
  free(descr);

  /* Free layout object */

  rc = m_destroy_layout(plh);
  if (rc)
     perror("m_destroy_layout : ");
}
```

The output of *m_transform_layout*( ) is the original string with some embedded escapes:

```
   inpbuff:  AB cde 123Z
```

Because the original orientation is LTR, the level 1 segment has an RTL orientation.

```
   outbuf:   <LTR>AB <Push><RTL>cde <LTR>123<Pop><LTR>Z
```

Example A-3 Explicit to Visual Buffer

The next program segment converts the explicit buffer to a visual buffer:

```
main
{
  LayoutObject plh;
  int rc=0;
  LayoutValues layout;
  LayoutTextDescriptor descr;
  int index;
  char *inpbuff="<LTR>AB <Push><RTL>cde <LTR>123<Pop><LTR>Z";
  char *outbuff;
  size_t inpsize;
  size_t outsize;
  size_t index=0;
  size_t query_size;

  /* Create layout object */

  plh = m_create_layout(NULL, NULL);

  layout=malloc(2*sizeof(LayoutValueRec));
  descr=malloc(sizeof(LayoutTextDescriptorRec));

  /*  Overwrite some of the locale default text descriptor settings.
  /*  But beforehand, query values of shapecharset.
```

```
 */
layout[0].name = ShapeCharset|QueryValueSize;
layout[0].value = &query_size;
       m_getvalues_layout(plh, layout, &index);

layout[0].name = ShapeCharset;
layout[0].value = (char *)malloc(query_size);
   rc = m_getvalues_layout(plh, layout, &index);
       if (rc)
       {
         perror("m_getvalues_layout : ");
          exit(0);
       }

layout[0].name=TypeOfText|TextShaping|Numerals;
layout[0].value=(caddr_t)descr;
layout[1].name=0;

descr->in=TEXT_EXPLICIT|TEXT_NOMINAL|NUMERALS_NOMINAL;
descr->out=TEXT_VISUAL|TEXT_SHAPED|NUMERALS_NATIONAL;

rc = m_setvalues_layout(plh,layout,&index);
if (rc)
{
    perror("m_setvalues_layout : ");
    exit(0);
}

/* Set the input buffer size and Allocate the output arrays */

inpsize = strlen(inpbuff);

outbuff = (char *)malloc(inpsize*sizeof(char));

rc = m_transform_layout(plh, inpbuff, inpsize, outbuff, &outsize,
                        NULL, NULL, NULL, &index);

if (rc)
{
    perror("m_transform_layout : ");
    exit(0);
}

/* user defined function that prints the output arrays to the user */

free(layout);
free(descr);
free(outbuff);

/* Free layout object */

rc = m_destroy_layout(plh);
```

```
  if (rc)
     perror("m_destroy_layout : ");
}
```

The output of the transform function is:

```
  AB 456 zyxZ
```

where 456 are the national numbers for 123, and zyx are the national letters for edc.

Appendix B

LO_LTYPE Locale Category

OP This appendix describes the LO_LTYPE category, which is in many respects an extension to the existing LC_CTYPE locale category.

LO_LTYPE can be implemented as part of the layout object, or its keywords may be added in the future to the existing locale categories.

The keywords define character classifications, mappings and character attributes. These are used by some of the reordering and shaping algorithms embedded in the *m_transform_layout*( ) and *m_wtransform_layout*( ) functions. In the following descriptions there are references to lists of such characters in appropriate standards. These lists are quoted for illustration only and do not imply dependence on a specific encoding scheme.

B.1 Character Classifications Related to Directionality

left_to_right
Left-to-right directionality. For example the letters A, B, C, D ... Z have a left-to-right directionality.

right_to_left
Right-to-left directionality. For example the letters of the Hebrew alphabet have a right-to-left directionality.

num_terminator
Numeric terminator required by the directional algorithms. For example in complex-text languages, the dollar sign or plus sign could be identified by the directional algorithm as numeral terminators.

num_separator
Separators of numerals of the portable character set (but not of national numerals). The term *numerals of the portable character set* is used to indicate numbers represented with the digits 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, which are part of the POSIX portable character set. An example of **num_separator** is the period, which can be used to separate numbers represented with Arabic digits (0, 1, 2 ... 9), but is not used with Hindi numbers (to avoid confusion with the value zero represented in Hindic digits by a period).

common_num_separator
Numbers separators both for the numerals of the portable character set and for national numerals. For example, a colon can be identified by a directional algorithm as a number separator both for Arabic numbers as well as for Hindi numbers.

segment_separator
Characters to be identified by a directional algorithm as segment separator characters. A segment is a portion of text, in general shorter than one line, embedded within a wider text that has a different directionality.

block_separator
Characters to be identified by a directional algorithm as 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.

B.2 Character Classifications of Control Characters

direction_control
Characters to be classified as direction control characters, such as those listed in the ISO/IEC 10646 standard. Examples of direction control are: Start Directed String (SDS) and Start Reversed String (SRS).

sym_swap_layout
Characters to be classified as symmetrical swap layout characters, such as those listed in the ISO/IEC 10646 standard. Examples of symmetrical swap layout characters are INHIBIT SYMMETRIC SWAPPING and ACTIVATE SYMMETRIC SWAPPING.

char_shape_selector
Characters to be classified as character shaping selectors, such as those listed in the ISO/IEC 10646 standard. Examples of character shaping selectors are INHIBIT ARABIC FORM SHAPING and ACTIVATE ARABIC FORM SHAPING.

num_shape_selector
Characters classified as numeric shaping selectors, such as those listed in the ISO/IEC 10646 standard. Examples of numeric shaping selectors are NATIONAL DIGIT SHAPES and NOMINAL DIGIT SHAPES.

B.3 Character Classifications of National Numbers

national_number
Characters to be classified as national numbers. Examples are 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.

B.4 Character Classifications of Composite Graphic Symbols

non_spacing
Characters to form composite graphic symbols, such as a character representing a diacritical mark in the ISO/IEC 6429 standard, or tone-marks, upper-vowels and lower-vowels in Thai.

B.5 Mapping Keywords

tosymmetric

This operand consists of character pairs, separated by semicolons. The characters in each pair are separated by a comma, and the pair is surrounded by parentheses. The first character of each pair is to be swapped with the second one. Symmetric characters are listed in the ISO/IEC 10646 standard. Examples are RIGHT and LEFT PARENTHESIS, GREATER THAN and LESS THAN signs, and so on.

tonational

This maps to national digits. The operand consists of character pairs, separated by semicolons. The characters in each 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

This operand consists of character pairs, separated by semicolons. The characters in each 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.

B.6 Character Classification Related to Character Connectivity

Normal_connect
: Characters that connect both to the left and to the right. This applies, for example, to many Arabic characters.

R_connect
: Characters that connect only to characters to their right and not to the left. In Arabic, for example, this includes characters like the reh, dal, waw, all the lamalefs, and the alef maksoura.

No_connect
: Characters that do not connect to the characters neither to their left nor to their right and cannot be overridden. For example, all the Latin characters, the box characters and the punctuation marks.

No_connect-space
: These are neither left nor right connectors, but they may be overridden if a neighbouring character needs to be expanded. For example, in Arabic these are the space, RSP and tail.

Vowel_connect
: All the connectable vowels. Vowels do not influence connectivity, but they need special consideration in scripts such as Arabic.

Special1
: Characters that need special handling. In Arabic, the Lam character when followed by Alef will form the ligature LamAlef, provided that no character of a class different than the Vowel class falls in between.

Special2
: Characters that need special handling. In Arabic, the Alef character when preceded by Lam will form the ligature LamAlef.

Special3
: Characters that need special handling. In Arabic, Seen, Sheen, Sad and Dad when displayed on two cells.

Special3 may differ for scripts of different languages.

Appendix C

Dynamic Pluggable Interface

OP This appendix defines the interface between the locale-independent and locale-dependent layer of Layout Services.

The locale-independent module is a library containing the interfaces defined in this document. These interfaces are implemented as wrapper functions which access the actual locale-specific implementation through function pointers. In addition, this locale-independent layout services library also contains the platform-specific implementation for dynamic loading of the locale-dependent module.

Each locale will have a locale-dependent module containing the implementation of the locale-sensitive functions. The locale-dependent module could be implemented as part of the locale-independent layout services library, or it could be implemented as a separate dynamic loadable object to be loaded in during runtime. This proposal will only concentrate on defining the dynamic loadable object approach.

The advantages to separate the implementation into locale-dependent and locale-independent modules are:

1. Each locale-dependent module could be highly tuned for a specific language to yield maixmum performance.
2. A client can switch to a different Layout Services support dynamically during runtime.
3. A single application can have a different Layout Services support for each window.

C.1 Operating System Requirement

In order to implement the dynamic loadable object approach, the underlying operating system must support the dynamic linking feature. This feature could be provided via the following two interfaces:

- an interface to load in a shared object library dynamically
- an interface to access a function within the share object library via a specified symbol.

C.2 Design

Every locale-dependent module must implement a function called *_LayoutObjectInit*( ). This is the entry point from locale-independent module to locale-dependent module to create and initialize **LayoutObject**.

Following is an example of how a locale-dependent module implements only the default Layout Services Support:

```
#include <sys/layout.h>

LayoutObject
_LayoutObjectInit(locale_name)
    char  *locale_name;
{
    return(_LayoutObjectDefaultInit(locale_name));
}
```

The locale-independent module has a built-in default implementation of the locale-dependent methods which can provide at least ‘‘C’’ locale support.

The naming of the libraries for locale-independent and locale-dependent modules are implementation-dependent. Following is an example for the location and naming of the libraries. The locale-independent module is **/usr/lib/liblayout.so**. The locale-dependent module is **$LOCPATH/<locale>.layout.so.<version number>**.

C.3 Data Structure

```
typedef struct _LayoutObject *LayoutObj;
                         /* Private name of Layout Object */
typedef struct _LayoutObject *LayoutObject;
                         /* Public name of Layout Object */

typedef struct {
    LayoutObject (*create)(
#if NeedFunctionPrototypes
        LayoutObj, AttrObj, LayoutValues
#endif
    );

    int (*destroy)(
#if NeedFunctionPrototypes
        LayoutObj
#endif
        );

    int (*getvalues)(
#if NeedFunctionPrototypes
        LayoutObj, LayoutValues, int *
#endif
        );

    int (*setvalues)(
#if NeedFunctionPrototypes
        LayoutObj, LayoutValues, int *
#endif
        );

    int (*transform)(
#if NeedFunctionPrototypes
        LayoutObj, const char *, size_t *, void *,
        size_t *, size_t *, size_t *, unsigned char *, size_t *
#endif
        );

    int (*wcstransform)(
#if NeedFunctionPrototypes
        LayoutObj, const wchar_t *, size_t *, void *,
        size_t *, size_t *, size_t *, unsigned char *, size_t *
#endif
        );

} LayoutMethodsRec, *LayoutMethods;

typedef struct {
    char   *locale_name;

    LayoutTextDescriptor orientation;
    LayoutTextDescriptor context;
    LayoutTextDescriptor type_of_text;
    LayoutTextDescriptor implicit_alg;
    LayoutTextDescriptor swapping;
    LayoutTextDescriptor numerals;
    LayoutTextDescriptor text_shaping;
    BooleanValue  active_dir;
    BooleanValue  active_shape_editing;
    char   *shape_charset;
    int    shape_charset_size;
    unsigned long  in_out_text_descr_mask;
    unsigned long  in_only_text_descr;
    unsigned long  out_only_text_descr;
    LayouDesc   check_mode;
    LayoutEditSize  shape_context_size;
} LayoutCoreRec, *LayoutCore;
```

```
typedef struct _LayoutObject {
    LayoutMethods methods; /* methods of this CTL Object */
    LayoutCoreReccore; /* data of this CTL Object */
    void  *private_data; /* Private data of locale-dependent object */
} LayoutObjectRec;
```

C.4 Calling Sequence

1. Within *m_create_layout*( ), *_LayoutObjectGetHandle*( ) is called to retrieve a **LayoutObject** for the specified locale by loading in the locale-dependent module.

2. In order to avoid the same locale-dependent module to be reloaded, a list of loaded locales is searched.

3. If the specified locale is found in the list, *_LayoutObjectInit*( ) will be called to create a new **LayoutObject** via the pointer within the locale-specific object.

4. Otherwise, *_GetLayoutInitFunc*( ) will be called to load in the locale-dependent module and return a function pointer to *_LayoutObjectInit*( ) for the specified locale. This function pointer will be added to a list to be reused for the same locale in the future as in step 2.

5. If the locale-dependent module cannot be loaded for the specified locale, *_LayoutObjectGetHandle*( ) will return NULL.

6. The actual creation and initialization of the **LayoutObject** is done within each locale-dependent implementation of *_LayoutObjectInit*( ).

C.5 Sample Code

```
/* Following is part of the locale-independent module */

#define CTLObjectRelease          1

static LayoutMethodsRec _DefaultLayoutMethods = {
 _LSDefaultCreate,
 _LSDefaultDestroy,
 _LSDefaultGetValues,
 _LSDefaultSetValues,
 _LSDefaultTransform,
 _LSDefaultWCSTransform,
    };

    };
typedef struct _InitFuncListRec  *InitFuncList;

typedef struct  _InitFuncListRec {
    void  *func   /* function pointer to _LayoutObjectInit() */
    char  *locale_name; /* locale of _LayoutObjectInit() */
    InitFuncList   next;  /* Pointer to next element of the list */
} InitFuncListRec;

/*
 *  Entry point of to create and initialize the default layout object.
 *
 */
LayoutObject
_LayoutObjectDefaultInit(locale_name)
    char  *locale_name;
{
    LayoutObj            layout_obj;

    /* Allocate layout_obj. */
    /* Initialize layout_obj->methods and layout_obj->core  */
    /* Return layout_obj. */
}

/*
 *  Search if a layout object of the requested locale is created.
 *  If not, create one and add it to the list.
 */
static LayoutObject
_DoGetObjHandle (locale_name)
    char   *locale_name;
{
    LayoutObj   layout_obj;

   /*
    * Look up a list of existing loaded locale specific objects.
    * If the locale specific object is already loaded for the specified
    * locale, call _LayoutObjectInit() to create the layout object
    * via the pointer within the locale object.
    */

   /*
    *  If the locale specific object is not loaded for the specified
    *  locale, then call _GetLayoutInitFunc() to load the
    *  locale-specific object.
    */

   /*
    * If _GetLayoutInitFunc() returns is not NULL, then add
    * the new object to the list and call _LayoutObjectInit() to
    * create layout_obj.  Then return layout_obj when its done.
    * Otherwise, return NULL.
    */
}

/*
 * Get LayoutObj of current locale.
 * If that failed, try it with C locale.
 */
LayoutObj
_LayoutObjectGetHandle (locale_name)
    char *locale_name;
{
    LayoutObj            layout_obj;

    /* Default locale_name to C if it is not set. */
    /*
     *  Use _DoGetObjHandle() to retrieve layout_obj
     *  base on locale_name and return layout_obj.
     */
}

/*
 *  Do actual loading of the load-dependent object.
 *  Then return the function pointer to _LayoutObjectInit.
 */
static void *
_GetLayoutInitFunc(locale_name)
    char  *locale_name;
{
    /* If locale_name is NULL, then return NULL. */
    /*
     * Otherwise, dynamically load in the locale specific
     * module.  Then return the function pointer to _LayoutObjectInit.
     */
}
```

C.6 Common Naming for Layout Values

```
#define AllTextDescriptors     0x0000007f

#define Orientation            1L
#define Context               (1L<<1)
#define TypeOfText            (1L<<2)
#define ImplicitAlg           (1L<<3)
#define Swapping              (1L<<4)
#define Numerals              (1L<<5)
#define TextShaping           (1L<<6)

#define ActiveDirectional     (1L<<16)
#define ActiveShapeEditing    (1L<<17)
#define ShapeCharset          (1L<<18)
#define ShapeCharsetSize      (1L<<19)
#define ShapeContextSize      (1L<<20)
#define InOutTextDescrMask    (1L<<21)
#define InOnlyTextDescr       (1L<<22)
#define OutOnlyTextDescr      (1L<<23)
#define CheckMode             (1L<<24)
#define QueryValueSize        (1L<<25)

/* Possible values for Orientation */
#define ORIENTATION_LTR        0x00000001
#define ORIENTATION_RTL        0x00000002
#define ORIENTATION_TTBRL      0x00000003
#define ORIENTATION_TTBLR      0x00000004
#define ORIENTATION_CONTEXTUAL 0x00000005

/* Possible values for Context */
#define CONTEXT_LTR            0x00000010
#define CONTEXT_RTL            0x00000020

/* Possible values for TypeOfText */
#define TEXT_IMPLICIT          0x00000100
#define TEXT_EXPLICIT          0x00000200
#define TEXT_VISUAL            0x00000300

/* Possible values for ImplicitAlg */
#define ALGOR_BASIC            0x00001000
#define ALGOR_IMPLICIT         0x00002000

/* Possible values for Swapping */
#define SWAPPING_NO            0x00010000
#define SWAPPING_YES           0x00020000

/* Possible values for Numerals */
#define NUMERALS_NOMINAL       0x00100000
#define NUMERALS_NATIONAL      0x00200000
#define NUMERALS_CONTEXTUAL    0x00300000

/* Possible values for TextShaping */
```

```
#define TEXT_SHAPED             0x01000000
#define TEXT_NOMINAL            0x02000000
#define TEXT_SHFORM1            0x03000000
#define TEXT_SHFORM2            0x04000000
#define TEXT_SHFORM3            0x05000000
#define TEXT_SHFORM4            0x06000000
```

```
/* Possible values for CheckMode */
#define MODE_STREAM             0x00000001
#define MODE_EDIT               0x00000002
```

```
#define MaskAllTextDescriptors 0x0fffffff
#define MaskOrientation         0x0000000f
#define MaskContext             0x000000f0
#define MaskTypeOfText          0x00000f00
#define MaskImplicitAlg         0x0000f000
#define MaskSwapping            0x000f0000
#define MaskNumerals            0x00f00000
#define MaskTextShaping         0x0f000000
```

```
/* Mask for the Property parament of m_*transform() */
#define NESTLEVEL_MASK          0x0f
#define DISPLAYCELL_MASK        0x10
```

Glossary

Some of the terms defined below are from The American National Dictionary for Information Processing, by the Computer and Business Equipment Manufacturers Association, 1977. These terms are identified by the acronym ANSI following their definitions.

algorithm
A finite set of well-defined rules for the solution of a problem in a finite number of steps. For example, a full statement of an arithmetic procedure for evaluating *sin x* to a stated precision. (ANSI)

Arabic country
Any of the countries in which *Arabic script* is the predominant writing system. The countries include Algeria, Bahrain, Egypt, Iraq, Jordan, Kuwait, Lebanon, Libya, Morocco, Oman, Qatar, Saudi Arabia, Sudan, Syria, Tunisia, United Arab Emirates, and Yemen.

Arabic numerals
The characters 1, 2, 3, 4, 5, 6, 7, 8, 9, and 0. Contrast with *Hindi numerals*. See also **numbers** on page 87.

Arabic script
A *cursive script* used in Arabic countries. Other writing systems such as Latin, Japanese, and Hebrew have a cursive handwritten form, but usually are typeset or printed in discrete letter form. Arabic script has only the cursive form.

Note: Arabic script is also used for Urdu (which is spoken in Pakistan, Bangladesh, and India), Farsi (or Persian, which is spoken in Iran, Iraq, and Afghanistan) and other languages that are not Arabic.

ascender
The parts of certain letters, such as b, d or f, which rise above the top edge of other letters such as a, c and e. Contrast with **descender** on page 84.

ASMO
Arab Standards and Metrology Organization.

AttrObject
AttrObject is a generic object which can be a container of many opaque objects. A locale is an example of the type of object that can be attached to the **AttrObject**. **AttrObject** is an object type other than an array type that can hold values that represent the locale-specific information necessary for all locale categories.

base shape
The form of an Arabic character that identifies it without specifying its presentation shape. See also **presentation shape** on page 88 and **shape determination** on page 88.

bidirectional languages
Languages such as Arabic, Hebrew and Yiddish whose general flow of text proceeds horizontally from right to left, but numbers, English and other left-to-right language text such as addresses, acronyms and quotations are written from left to right.

cell

A group of character elements that belong to the same composed character. Also called display cell.

character elements

The components of a composed character such as a ‘‘Thai Character’’, namely base line consonants, upper vowels, lower vowels, base line vowels, tone marks, diacritics, and so on.

charset

An encoding with a uniform, state-independent mapping from character to code points. Usually (but not necessarily), the code points are related to adequate presentation glyphs, that when presented use associated fonts.

complex-text languages

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

composed character

A collection of character elements in some scripts, such as Thai or Lao, whose presentation forms compose a glyph that occupies a definite space called a presentation cell. Also called combined character.

composing character element

A character element, such as a Thai tone mark, a Thai upper or lower vowel or diacritic, that together with the non-composing character element and possibly other composing-character elements forms a composed character. Sometimes called a composing character or a combining character. In a string of character elements it follows the non-composing character element. When presented the composing character element does not occupy, normally, a separate presentation cell, and it shares the same cell with a non-composing character element and possibly with other composing character elements.

composite sequence

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

control character

A character that denotes the start, modification, or end of a control function. A control character can be recorded for use in a subsequent action, and it can have a graphic representation.

cursive script

Script whose adjacent characters might touch or be connected to each other. Arabic, Farsi and Urdu scripts are always cursive, while Latin script is cursive only in handwriting.

data entry

The method of entering data into a computer system for processing, usually in a field-oriented environment where the entry is governed by a program.

descender

The part of the character that extends from the baseline to the bottom of the character cell. Examples of letters with descenders are g, j, p, q, y and Q. Contrast with **ascender** on page 83.

deshaping
The opposite of shaping; the transformation of an Arabic language text to a layout used for processing. The different shapes of the same character are *folded* into a single, *basic* shape.

diacritic
Modifying mark of a character. For example, the accent marks in Latin scripts (acute, tilde and ogonek), the vowel marks in Hebrew, and the consonant pronunciations in Thai and Lao.

display cell
The group of character elements that form a composed character.

enable (national languages)
To design a product for economical and easy adaptation to any culture, convention or language of the user.

encoding scheme
A set of specific definitions that describe the philosophy used to represent character data. The number of bits, the number of bytes, the allowable ranges of bytes, the maximum number of characters, and the meanings assigned to some generic and specific bit patterns, are some examples of specifications found in such a definition.

ECMA
(European Computer Manufacturers Association) A not for profit organisation formed by European computer vendors to promulgate standards applicable to the functional design and use of data processing equipments.

explicit algorithm
In a bidirectional text, an algorithm that identifies segments of different directionality, or other peculiarities of characters (such as shaping). The algorithm uses *explicit* control sequences (directional and other) embedded in the text. See also **implicit algorithm** on page 86.

field attributes
The data description governing the presentation and handling of data in the associated data field. For example, direction (left-to-right, right-to-left) is a field attribute important in bidirectional applications.

folding
The substitution of one graphic character for another. Folding generally maps a larger character set into a subset, and may result in some loss of information. For example, folding allows printing of upper-case graphic characters when lower-case characters are not available.

global orientation
The predominant orientation of a bidirectional text. For example, an Arabic text with a right-to-left global orientation may have some left-to-right English names embedded in it.

glyph
A member of a set of symbols that represent data. Glyphs can be letters, digits, punctuation marks, or other symbols.

graphic character
A member of a set of symbols that represent data. Graphic characters can be letters, digits, punctuation marks, or other symbols. Synonymous with *glyph*.

graphic symbol
The visual representation of a graphic character or of a composite sequence.

A graphic symbol for a composite sequence generally consists of the combination of the graphic symbols of each character in the sequence.

Hangul

The Korean alphabet that consist of fourteen basic consonants and ten basic vowels. Hangul was created by a team of scholars in the 15th century at the behest of King Sejong. See also **JAMO**.

Hanja

The Korean term for characters derived from Chinese.

Hindi numerals

A set of numerals used in many Arabic countries instead of or in addition to the ''Arabic'' ones. Hindi numeral shapes are:

٠ ١ ٢ ٣ ٤ ٥ ٦ ٧ ٨ ٩

which correspond to the Arabic numeral shapes of 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9. Contrast with **Arabic numerals** on page 83.

ideographic language

A written language in which each character represents a thing or an idea. An example of such a language is Chinese. Contrast with **phonetic language** on page 87.

implicit algorithm

An algorithm that recognises directional segments based on the *implicit* characteristics of the characters. Segments are inverted accordingly. Bidirectional text transformed using an implicit algorithm is stored in logical order. See also **explicit algorithm** on page 85 and **logical order (or logical sequence)** on page 87.

JAMO

A set of consonants and vowels used in Korean Hangul. The word JAMO (or jamo) is derived from ja, which means consonant, and mo, which means vowel. See also **Hangul**.

language layer

A keyboard may have several language layers. For example, the Hebrew keyboard may have three layers: Hebrew, English and APL, with each layer supporting up to three shifts (lower-case, upper-case and alternate shifts).

Latin alphabet

An alphabet comprising the letters a, b, c, d, e, f, g, h, i, j, k, l, m, n, o, p, q, r, s, t, u, v, w, x, y, and z in upper-case and lower-case, with or without accents and ligatures. Contrast with **non-Latin-based alphabet** on page 87.

layout

In this document *layout* stands for the *layout* of a text: the direction of the segments and the shape of the characters.

LayoutObject

An opaque object containing all the data and methods necessary to perform the layout operations on context-dependent or directional characters. In particular it contains a set of layout values.

layout transformation

A transformation between the layout of a text as processed and the layout of text when presented. A layout transformation may involve determination of embedded directional segments, segment inversion, character shaping, or character deshaping.

layout value
A set of text attributes and processing indicators needed by the layout transformation functions. See also **text attribute** on page 89.

ligature
A graphic character consisting of two or more characters joined together. For example, joining A and E forms the ligature Æ. Ligatures are very common and important in Arabic.

logical order (or logical sequence)
A bidirectional text is said to be stored in logical order if the data elements in each segment are sequenced physically in keystroke order (order of entry): that is the order they would be read from a screen or spoken aloud. The segments presented with an opposite directionality to the global orientation, need to be inverted to be stored in logical order.

lower case
The small alphabetic characters, whether accented or not, as distinguished from the capital alphabetic characters. The concept of case also applies to alphabets such as Cyrillic and Greek, but not to Arabic, Hebrew, Thai and some other languages. Examples of lower-case letters are a, b and c. Contrast with **upper case** on page 89.

monocasing
The translation of alphabetic characters from one case (usually lower case) to their equivalents in another case (usually upper case).

national numbers
Numbers as written in a text of a language that has its own glyphs for digits. As an example, a Thai text may have numbers represented by national numbers called Thai numerals. Note that the national numbers in the Arabic languages are the Hindi numerals and not the *Arabic* numerals. See also **Hindi numerals** on page 86 and **Arabic numerals** on page 83.

nesting
The situation in which a directional segment is embedded within another directional segment. It is possible to have more than one level of nesting. A left-to-right number can be nested, for example, within a right-to-left Hebrew text, which itself is nested within a left-to-right English text.

non-composing character element
A character element such as a Thai consonant around which all the other character elements are composed. Sometimes called a non-composing character or a non-combining character. In a string of character elements it is the first element of a composed character. When presented the non-composing character element occupies, alone or with composing elements a presentation cell.

non-Latin-based alphabet
An alphabet that is not a Latin alphabet. Examples are Greek and Arabic alphabets. Contrast with **Latin alphabet** on page 86.

numbers
Numbers express either quantity (cardinal) or order (ordinal). Many cultures have different forms for cardinal and ordinal numbers. For example in French the cardinal number five is *cinq*, but the ordinal fifth is *cinquième or* 5^{eme} *or* $5^{è}$. *Numbers are written with symbols usually referred to as numerals. See* **Arabic numerals** *on page 83 and* **Hindi numerals** *on page 86.*

phonetic language
A written language in which one or more characters represent a sound. Examples of phonetic languages are English, Greek and Russian. Contrast with **ideographic language**

on page 86.

physical order (or physical sequence)
A bidirectional text is said to be stored in *physical order* if each data element of each segment of the text is stored in the same physical sequence that is presented.

presentation
Printing or displaying.

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.

presentation layout
The layout of text when presented on a screen or on a printer. See also **layout** on page 86.

presentation shape
The shape of a character such as an Arabic character when presented to the user. See also **base shape** on page 83 and **shape determination**.

processing layout
The layout of text when processed.

push mode
An operating mode for entering text in reversed orientation where the cursor remains stationary while new characters are typed, and the beginning of the text is *pushed* as in a pocket calculator. It is also called calculator mode.

right-to-left mode
An input mode of a bidirectional text in which the cursor moves to the left after the entry of each character.

segment
A contiguous portion of text with one directionality that may or may not be embedded in an other portion of text which has a different directionality. For instance, in a bidirectional text, a left-to-right segment such as a number can be embedded in a text that has a right-to-left directionality.

ShapeCharset
The charset used to shape text (ShapeCharset is not necessarily identical to the encoding of the text before the shaping).

shape determination
A process that decides which of the several (up to four) shapes an Arabic character is to be used in current context. The shapes are initial, middle, final, and isolated. For each character, the decision is based on the linking capabilities of current and surrounding characters. See also **base shape** on page 83 and **presentation shape**.

shaping
The process of presenting a cursive script text with characters properly shaped as initial, middle, final or isolated shape, according to their context. See **shape determination** and **cursive script** on page 84.

symmetrical swapping
The process of exchanging some characters (such as { or <) with their symmetric twin character (such as } or > respectively). The symmetrical swapping may be performed during the inversion of segments of a bidirectional text.

text-type (or TypeOfText)
Text-type is used to indicate which reordering approach (visual, implicit or explicit) applies to a given bidirectional text. See also **implicit algorithm** on page 86, **explicit algorithm** on page 85 and **visual data**.

text attribute
Text attributes such as text-type, compliance to symmetrical swapping, numerals shape or character shaping, describe the complex text being transformed. The text attributes are used by the layout transformation function. See also **layout transformation** on page 86 and **layout value** on page 87.

upper case
The capital alphabetic characters, whether accented or not, as distinguished from the small alphabetic characters. The concept of case also applies to alphabets such as Cyrillic and Greek, but not to Arabic, Hebrew, Thai and some other languages. Examples of capital letters are A, B and C. Contrast with **lower case** on page 87.

visual data
Visual data is composed of data elements that are sequenced in the same order that they are presented on a screen or printer.

Index