

Sun™ Software Product Internationalization Taxonomy

Contents

Chapter 1 Introduction

[1.1 Purpose of This Document](#)

[1.2 How to Use This Document](#)

[1.3 Key Terminology](#)

Chapter 2 Planning for Internationalization

[2.1 Planning Ahead](#)

[2.2 User Categories](#)

[2.3 Data Categories](#)

[2.4 Designing the Roadmap](#)

[2.5 Additional Considerations](#)

Chapter 3 Interfaces

[3.1 Command Line Interface](#)

[3.2 Character Interface](#)

[3.3 Graphical Interface](#)

[3.4 Application Protocols](#)

[3.5 Storage and Interchange](#)

[3.6 Application Programming Interfaces \(APIs\)](#)

Chapter 4 Objects and Methods

[4.1 Translatable Product Components](#)

[4.2 Cultural Formatting and Processing](#)

[4.3 Text Foundation and Writing Systems](#)

Chapter 5 Internationalization Assessment Matrix

Chapter 6 Reading the Matrix

[6.1 Audience](#)

[6.2 The Matrix at a Glance](#)

[6.3 What Do the Boxes Represent?](#)

[6.4 Relating the Matrix to the Product](#)

[6.5 Potential Showstoppers](#)

[References](#)

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

Chapter 1 Introduction

1.1 Purpose of This Document

This document defines what it means for a software product to be internationalized. It is intended for:

- **Software designers** - To evaluate internationalization areas as they pertain to their product designs
- **Product review committee members** - To determine if a product complies with internationalization policy
- **Engineers** - To understand internationalization considerations in implementing a product design
- **Engineering managers** - To aid in determining the product road map for resource planning
- **Software quality engineers** - To define test plans and create tests that verify adequate internationalization
- **Technical product reviewers** - To aid in evaluating the internationalization status of a product

The benefit behind defining internationalization is having a common understanding of the term "internationalized", which helps to communicate product information throughout the company. A further benefit is better integration and interoperability of products. Customers will see consistent functionality from one product to another, and will buy additional products with the confidence that new products will handle their data in the same manner as those they already use. This document does not describe how to internationalize a product. While there are some coding examples given to illustrate the concepts described in the text, they are not intended as a guide to implementation. Please see the ["References"](#) section for further reading.

1.2 How to Use This Document

This document describes the [Internationalization Assessment Matrix](#). The matrix is an overview of internationalization in different areas of a software product. The columns in the matrix represent different types of interfaces that a product can provide for users or other software products. The rows represent different objects and areas of functionality that a software product might offer. The matrix lets you quickly determine which areas are

relevant to your product and which ones are not.

Begin by reading Chapters 1-3 of this document to gain an understanding of the terminology. Then examine the matrix along with the related descriptions in Chapter 4. The "Descriptions" and "Requirements for Compliance" sections are structured to correspond to the boxes in the matrix. Determine the relevant cells for your product, and address the requirements for these cells.

1.3 Key Terminology

A familiarity with key terms is necessary for understanding the rest of this document.

Internationalization (I18n)

The word internationalization is sometimes abbreviated as i18n, because there are 18 letters between the *i* and the *n*. There are several definitions of internationalization in books and on Web sites. In this document, internationalization is making program code generic and flexible so that specifications from markets around the world can be easily accommodated. Part of internationalization is making the product localizable, that is, enabling the interface to be translated with a minimal impact on the program code.

For example, instead of hard-coding message text, as in:

```
/* This is not internationalized code */
printf("This message needs internationalization.");
```

the message is externalized into a message catalog, using `catgets()`:

```
/* This is internationalized code */
printf(catgets(mycatalog, 1, 1, "This message is now
internationalized."));
```

Another important part of internationalization is allowing the data processing to handle characters and data formats from all over the world.

For example, instead of assuming a sort by byte value, as in this C fragment:

```
/* This is not internationalized code, since strcmp()
compares byte
values, producing an invalid sort order */
if (strcmp((const char *)string[q], (const char *)string[p])
> 0)
{
    temp = string[q];
    string[q] = string[p];
    string[p] = temp;
}
```

The strings are sorted by locale conventions using a locale-sensitive function:

```
/* This is internationalized code, since strcoll() uses
locale
```

```

information to determine sort order */
if (strcoll((const char *)string[q], (const char *)string[p])
> 0)
{
    temp = string[q];
    string[q] = string[p];
    string[p] = temp;
}

```

Internationalization comes in several variants:

- **Monolingual internationalization** - Enables the creation of localized versions of a product, where each localized version supports only its target locale. This is no longer sufficient for business requirements.
- **Internationalization for multilocalization** - Supports localization and data processing for multiple locales, where the actual locale is selected on execution of the product or at runtime.
- **Multilingualization** - Enables data processing and display in multiple languages and locales simultaneously, for example, mixing Chinese and Arabic text in a single document.

Internationalization for multilocalization is the minimal requirement, but many products should be designed to be fully multilingual. Server software in particular needs to be able to handle user requests in different languages and with different cultural conventions simultaneously.

Localization (L10n)

Sometimes you will see localization abbreviated as l10n, because there are 10 letters between the *l* and the *n*. Localization is the process of customizing a product for a particular locale. This can involve several different types of customization, including:

- Translation of the user interface and documentation into a different language.
- Altering some format fields in resource files according to the locale conventions, for example, changing a date format from mm/dd/yy to yy/mm/dd.
- Adding code modules that implement locale-specific functionality, such as an input method editor for Japanese or a module that calculates Hebrew calendar dates.

Not all internationalized products are localized. This document is concerned with localization only to the extent that it defines requirements for internationalization.

Locale

A locale is a specific geographical, political, or cultural region. It is usually identified by a combination of language and country, for example, en_US represents the locale US English. In some cases, a language identifier alone is sufficient.

Providers and Consumers

This document divides software functions into two groups: providers and consumers. A

software product usually has both provider and consumer functionality, depending on the product area; therefore, each combination of an interface and an object in the matrix has a provider and a consumer.

A provider provides internationalization functionality. For example, an input method is a provider of internationalized input functionality for an application programming interface (API). It contains calls and parameters that a consumer can use to retrieve data typed in by the user, or to display the input method editor window. For example, to ensure that international data is handled correctly, a consumer uses parameters, such as character set name and locale in a provider application protocol.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

References

del Galdo, Elisa M., and Jakob Nielsen, *International User Interfaces*, New York: John Wiley and Sons, 1996, 288 pages, ISBN: 0-471-14965-9.

Fernandes, Tony, *Global Interface Design*, Chestnut Hill, MA: Academic Press, 1995, 191 Pages, ISBN: 0122537904.

O'Donnell, Sandra Martin, *Programming for the World: A Guide to Internationalization*, Prentice Hall, April 1994, 440 pages, ISBN: 0-13-722190-8.

Rhind, Graham, *Global Sourcebook of Address Data Management: A Guide to Address Formats and Data in 194 Countries*, Gower, 1999, 640 pages, ISBN: 0-566-08109-1.

Rubric, *Tips for Creating World-ready Documentation*, [<http://www.rubric.com>].

Uren, Emmanuel, Robert Howard, and Tiziana Perinott, *Software Internationalization and Localization: An Introduction*, Van Nostrand Reinhold, June 1993, 300 pages, ISBN: 0442014988.

Weider, et al., *The Report of the IAB Character Set Workshop*, RFC 2130, [<http://www.ietf.org/rfc/rfc2130.txt>], April 1997.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

Chapter 5 Internationalization Assessment Matrix

Description

This checklist is in the form of a matrix that enables you to assess the internationalization status of a product. Across the x-axis are the categories known as interfaces. For more information, see "[Chapter 3 Interfaces](#)" or the link at the top of each column in the matrix. Down the y-axis are the categories known as objects and methods. For more information, see "[Chapter 4 Objects and Methods](#)" or the link on each row in the matrix.

Each available box in the matrix should be populated with one of the following color-coded values:

- **Compliant** - Product fulfills all the requirements defined for this interface/object combination.
- **Partially compliant** - Product fulfills some of the requirements defined for this interface/object combination. An accompanying explanation in text form should appear after the matrix.
- **Non-compliant** - Product does not fulfill any of the requirements defined for this interface/object combination. Plans for future inclusion should be provided in a roadmap.
- **Not applicable** - Product does not provide the functionality of this interface/object combination.

To see a sample matrix, which we have already completed for demonstration purposes, [click here >>](#).

To use the matrix template to assess the internationalization status of your product, [click here >>](#).

		User Interfaces			Program Interfaces		
		Command line	Character	Graphical	Application Protocols	Storage and Interchange	Application Programming
1. Translatable Product Components	1.1 Translation negotiations, defaults, and selection						
	1.2 Textual objects	1.2.1 Fixed textual objects					
		1.2.2 Messages					
		1.2.3 Help systems and documentation					
	1.3 Non-textual objects	1.3.1 Icons, images, and colors					
		1.3.2 GUI objects					
		1.3.3 Sounds					
		1.3.4 Other					
	2. Cultural formatting and processing	2.1 Culture negotiations, defaults, and selection					
		2.2 Abstract objects	2.2.1 Time, date, and calendar				
2.2.2 Numeric, monetary, and metric							

	2.3 Structured text	2.3.1 Ordered lists (collation)							
		2.3.2 Personal names, honorifics, and titles							
		2.3.3 Addresses							
		2.3.4 Other formatting and layout							
	2.4 Language processing	2.4.1 Lexical and grammatical							
		2.4.2 Phonology and sound-to-text							
3. Text foundation and writing systems	3.1 Writing system negotiations, defaults, and selection								
	3.2 Plain text representation	3.2.1 Characters (semantics and codespaces)							
		3.2.2 Strings (encoding methods and transcoding)							
	3.3 I/O and interchange	3.3.1 Transfer encoding (8-bit clean)							
		3.3.2 Device input (keyboard and input methods)							
		3.3.3 Device output (font management, rendering, and output methods)							

Explanations for partially compliant areas and notes:

Important dependencies:

Location of product roadmap for providing future i18n functionality:

Chapter 3 Interfaces

This chapter divides possible software interfaces into groupings for the purpose of discussion. A project team should use these groupings when describing the internationalization issues and techniques employed for the interfaces of a software project. The objective is to allow generalizations for describing the internationalization issues of a whole class of interfaces at once. Most software projects have an interface in more than one of the following sections, but software projects are unlikely to have an interface in every section.

Whether a certain interface belongs in one grouping or another might not be completely clear. If this is the case, the interface can be included in whatever section eases the discussion of that interface and its internationalization issues.

3.1 Command Line Interface

The command line interface refers to the shell command used to start a program, including both the command name and its arguments and options.

NOTE: The Solaris guideline for command lines, which is based on based on `getopt` rules, specifies that option names must be one character long. Using single character names for options can aid international use, as non-English speakers should find it easier to type `-n` than `-newer`. For more information, type the following command:

```
man -s 1 intro
```

On the man page, see the "Command Syntax Standard: Rules" section.

We also group a command line interpreter in this section. The command line interpreter refers to any program that acts on subcommands typed by a user, for example, the shell and `dbx`.

3.2 Character Interface

A character interface is a user interface which is character or text based; that is, non-graphical, for example:

- Full-screen interface, such as the `vi` or Emacs text editors
- Output to a daisy wheel or other character-only printer
- Pop-up description box

Certain types of terminal messaging also fall into this category.

3.3 Graphical Interface

A graphical interface renders information onto a bit mapped (pixelated) or vector drawn space. Some examples are:

- Graphical user applications, such as a word processor or web browser.
- Output to a PostScript or PCL printer, or to a phototypesetter.
- Drawing on a pen plotter.

NOTE: Many graphical applications behave as if they were character interfaces. They use the default fonts supplied by the window system, assume fixed width characters, and use no icons or other graphic elements.

3.4 Application Protocols

An application server provides a set of well-defined services to client applications using some form of interprocess communication. The protocols used are interfaces and are often layered on lower-level protocols. Protocols are sometimes exposed to customers, so software on other systems can communicate with local software as a server, client, or peer.

For example, a Simple Mail Transfer Protocol (SMTP) server accepts email for delivery on a known communication port. Most email messages are intended for humans, even if produced programmatically, for example, by cron, the UNIX® clock daemon. However, some messages might be communicating with a server of some kind that understands the format of messages directed to it. In this situation, SMTP is being used as a store-and-forward transmission medium. Both the lower-level (SMTP on TCP/IP) and higher-level protocols (the programmatic message content) are protocol interfaces. They have separate syntax and semantics descriptions, and hence potentially different internationalization choices.

Private protocols are sometimes called messages or interprocess communication. For the purpose of internationalization, these are all grouped in this section; however, a private protocol might not require internationalization.

Server software and hardware is often shared by speakers of different languages. Protocol elements or data structures should be as universal as possible and interpreted for human consumption by internationalized and localized clients.

3.5 Storage and Interchange

File formats include the input formats accepted, output formats produced, and storage formats, whether documented for customers or kept private for internal use. Many file formats are a mixture of customer data and required syntax or "boilerplate."

File formats can be transported across a network to a different host. If appropriate, file formats should be reusable in various timezones, languages, and cultures. Since entire networks of systems cannot be upgraded simultaneously, different systems are likely to be

running different versions of any software; therefore, exchanging files leads to additional versioning considerations.

A database is the same concept, whether implemented in multiple files or not, and whether useful for interchange or solely for storage. Databases are therefore grouped in this section.

Logging, log files, and controlled file names and file suffixes are also assigned to this section.

3.6 Application Programming Interfaces (APIs)

APIs are programmer calling sequences to access a software library. When compiled, the library implementation and the client programs conform to an application binary interface (ABI). If they use the same translation rules, the ABIs are compatible and the client and library can communicate successfully.

Within an API, there might be aspects that require potentially different internationalization decisions. Function names, like command names, are not altered; they are the same worldwide for program and programmer portability. However, messages printed by a function should be internationalized. In general, API functions should not print error messages, but should provide status values to the caller.

Certain libraries also offer a systems programming interface that implementations can plug into. These implementations are often device-specific. They are also considered part of this section.

Chapter 4 Objects and Methods

This chapter includes the following sections:

4.1 Translatable Product Components

- [4.1.1 Translation Negotiations, Defaults, and Selection](#)
- 4.1.2 Textual Objects
 - [4.1.2.1 Fixed Textual Objects](#)
 - [4.1.2.2 Messages](#)
 - [4.1.2.3 Help Systems and Documentation](#)
- [4.1.3 Non-textual Objects](#)
 - [4.1.3.1 Icons, Images and Colors](#)
 - [4.1.3.2 Graphical User Interface \(GUI\) Objects](#)
 - [4.1.3.3 Sounds](#)
 - [4.1.3.4 Other Non-textual Objects](#)

4.2 Cultural Formatting and Processing

- [4.2.1 Culture Negotiations, Defaults, and Selection](#)
- 4.2.2 Abstract Objects
 - [4.2.2.1 Time, Date, and Calendar](#)
 - [4.2.2.2 Numeric, Monetary, and Metric](#)

- 4.2.3 Structured Text
 - [4.2.3.1 Ordered Lists \(Collation\)](#)
 - [4.2.3.2 Personal Names, Honorifics, and Titles](#)
 - [4.2.3.3 Addresses](#)
 - [4.2.3.4 Other Formatting and Layout](#)

- 4.2.4 Language Processing
 - [4.2.4.1 Lexical and Grammatical](#)
 - [4.2.4.2 Phonology and Sound-to-Text](#)

4.3 Text Foundation and Writing Systems

- [4.3.1 Writing System Negotiations, Defaults, and Selection](#)

- 4.3.2 Plain Text Representation
 - [4.3.2.1 Characters \(Semantics and Codespaces\)](#)
 - [4.3.2.2 Strings \(Encoding Methods and Transcoding\)](#)

- 4.3.3 I/O and Interchange
 - [4.3.3.1 Transfer Encoding \(8-Bit Clean\)](#)
 - [4.3.3.2 Device Input \(Keyboard and Input Methods\)](#)
 - [4.3.3.3 Device Output \(Font Management, Rendering, and Output Methods\)](#)

Copyright 1994-2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.1 Translatable Product Components

4.1.1 Translation Negotiations, Defaults, and Selection

Description

One of the key issues an application has to resolve is determining the target language for translatable product components. The method for doing this is usually platform dependent. On Solaris[tm], for example, it can be set either by the user or system administrator using environmental variables. Also, on Solaris it is usually set *globally*; that is, a user's entire session is set to the chosen language and cannot change without the risk of inadvertently affecting other processes in the same session. This is usually referred to as not being multithread (MT) safe. Java[tm] applications, on the other hand, allow users to change the target language for translatable components at any time without affecting other processes. This is achieved by treating a locale as an object that can be supplied with a call to any locale-sensitive operation. It is also important that if translated components for a selected language are not available, then an appropriate default mechanism should be used.

There is not always a one-to-one relationship between language and country, for example, Canada has two national languages: French and English. Switzerland has four: German, French, Italian, and Romansch. Chinese is really a family of dialects, many of which are mutually incomprehensible as spoken. There are also two distinct and mutually incomprehensible written forms: Traditional Chinese and Simplified Chinese. It is inappropriate to fall back from one to the other if one is unavailable. This is complicated enough when applied to a single platform situation. Imagine, however, trying to deal with this issue in a distributed system where client, server, and database might all be running on different machines and in different locales. How can one platform know what language another is using? These are typical problems faced by developers when developing properly internationalized software.

Default behavior is also important with regard to translation negotiations. There are two main reasons for this. Firstly, default behavior is required if translated components are not found. Secondly, most systems have default locations for translated components which means that application programming interfaces (APIs) do not need to specify full path names when accessing message catalogs, unless such catalogs reside at proprietary locations.

Command Line Interface

Environmental variables, which can be set on the command line, play an important part in determining how your application locates translatable components. Also, the mechanism that you use to access these translatable components determines which environmental variables are used. On Solaris, you can access translated components using several different mechanisms. Two of these are `gettext()` and `catgets()`. The former is a Sun proprietary method, the latter is a standard method from XPG4.

Solaris Example

`gettext()` retrieves translated text from a message catalog whose location is determined by a combination of locale name (established using environmental variables and a call to `setlocale()`), a call to `textdomain()`, and an optional call to `bindtextdomain()`.

The location is usually of the form:

```
.../locale/LC_MESSAGES/domain.mo
```

where:

locale is the current locale.

domain is the argument passed to `textdomain()`, which is effectively the filename of the message catalog.

Message catalogs usually have a default location, for example, `/usr/lib/locale/locale/LC_MESSAGES/domain.mo`. To change this location, use `bindtextdomain()`. To establish the current locale, use `setlocale()`, which uses environmental variables. The following table details precedence rules that are used to determine the locale.

Table 4-1. Precedence Rules for Environment Variables

Precedence	Variable
Highest	LC_ALL
Middle	LC_CTYPE
	LC_TIME
	LC_MONETARY
	LC_NUMERIC
	LC_COLLATE
	LC_MESSAGES
Lowest	LANG

NOTE: These environment variables are recognized by most UNIX platforms.

UNIX Example

`catgets()` retrieves translated text from a message catalog whose location is determined by a combination of locale name (established using environmental variables and a call to `setlocale()`), a call to `catopen()`, and a special environment variable called `NLSPATH`. `NLSPATH` is a series of colon (:) separated paths which can contain wildcard characters with special meanings. The following is an example of `NLSPATH` usage:

```
/usr/lib/locale/%L/LC_MESSAGES/%N.cat:/tmp/%N.%L.cat
```

In this example, `catopen()` replaces `%N` with its first argument; that is, the name of the file that contains the translated messages, minus the `.cat` suffix. `%L` is the full locale name as established by `setlocale()`. So, if the locale were set to `fr_FR.ISO8859-1`, `catopen()` would use the following file to retrieve French messages:

```
/usr/lib/locale/fr_FR.ISO8859-1/LC_MESSAGES/filename.cat
```

If this file does not exist, `catopen()` tries the following path:

```
/tmp/filename.fr_FR.ISO8859-1.cat
```

If `NLSPATH` is not set, a default location is used. The default location is usually:

```
/usr/lib/locale/locale/LC_MESSAGES/filename.cat
```

Other wildcard characters can be used here also. These can be platform dependent so you should check the platform documentation for these. The following table summarises the environment variables used with `gettext()` and `catgets()`, the two methods for retrieving text.

Table 4-2. Relationship Between Environment Variables and APIs for Translation

Negotiation

Function	Standard	Locale establishment	Catalog filename establishment	Catalog directory name establishment
gettext()	Sun	LC_MESSAGES/setlocale()	textdomain()	default or bindtextdomain()
catgets()	XPG4	LC_MESSAGES/setlocale()	catopen()	default or NLSPATH/catopen()

Some toolkits also allow you to specify the locale on the command line when invoking the command. The X Toolkit (Xt) Intrinsic layer, for example, provides a resource for this called XnLanguage. Two examples of how this resource can be used include:

- **Using a command line option** - Type the following command:

```
myapplication -xnllanguage ko
```

- **Using an X11 resource file** - You can add the following line to your .Xdefaults file:

```
*xnllanguage:      zh_TW
```

Character Interface

In a user interface (UI), translation comes into play in two areas: The interface itself is presented in one or more languages. Through the interface, users request or enter data for the software to process. In both cases, the interface can supply the user with language options. For example, a product that is shipped with multiple translations can enable the user to select a default that will be displayed each time the program is executed. In addition, the interface can supply a menu that enables the user to select different languages, resulting in a dynamic change to the UI text. Sometimes, additional fields or options are offered depending on the language.

When the user enters data into the system, a method too indicate the input language is required. The program might require the data to be in the same language as the interface, or it might provide an option to set the language of the input data.

Graphical Interface

For information on text in the UI, see ["Character Interface."](#)

In some cases, graphical elements can be used to select different languages. For example, flags are often used for language selection, though this is not the best graphical element as it is country-biased and may cause offense. Sometimes the graphics are simply the text name of the language in that language. This is difficult to display in text mode because the character and font support are usually not available for all the languages on the same page. Also, graphical elements can change along with text, or additional graphics can appear in certain languages.

Application Protocols

Translation negotiation is particularly important among different applications that are running either on different platforms or on the same platform. It becomes even more complicated if those

platforms are not homogeneous. Imagine, for example, a client application running on one platform requesting information from a server application on another whose database resides on yet another machine. Assume also that each application on each platform is running on a different locale. If an error message is generated from the database, what will eventually be displayed on the client end? This is the type of issue that a distributed international application needs to address.

Storage and Interchange

Files need to be stored in a way that enables desired translations to be retrieved. In the [UNIX example's](#) "Command Line Interface" section, the catalog files accessed by `catgets()` can be retrieved according to their file name. Java resource bundles have a language or locale as part of the class file name. Storing a file with a language tag as part of the file name can be invaluable for files that are exchanged or moved frequently. Using standard language or locale extensions enables the programmatic building of a language-specific file name.

Application Programming Interfaces (APIs)

Environment variables and certain APIs work closely in negotiating the target language for an application. For more information, see the ["Command Line Interface"](#) section.

NOTE: If you use a toolkit, it might not be necessary to call `setlocale()` directly. The toolkit might do it for you. For example, the Xt toolkit must be initialized to handle international data. In Xt, this is normally done using the `XtSetLanguageProc()` call. This is how Motif applications, for example, set their locale. `XtSetLanguageProc()` actually calls `setlocale()`.

Requirements for Compliance

Command Line Interface

If your application is written using a toolkit that supports a locale command line option or resource, for example, Xt, then that application automatically complies with the recommendations in this section.

NOTE: Not all toolkits or libraries provide this support.

Also, if your application uses the appropriate APIs for translation negotiation, the application does not need to change if environment variables change. It is also important that your application can fall back to some default behavior if it is unable to establish a target locale.

Character Interface

The interface must provide users with a clear choice for language display and language data processing. For a monolingual application, the data coming in from the user and the data returned should be in a language expected by the user. In other words, it must be clear to the user what data they are allowed to enter and what language the program assumes it to be. The language of the data that the application returns should also be obvious to the user.

In multilingual applications, language options must be clearly presented. Users must be able to understand what happens when they select a language option. For example:

- Does the language of interface elements change immediately or after they restart the application?
- Does the application assume that the language of all subsequent input data is the selected language?

- Is all subsequent data that the application returns in the selected language?

Graphical Interface

See the ["Character Interface"](#) requirements.

Application Protocols

You need to ensure that your application properly negotiates its target locale--and hence language--in situations where application components are running on the same platform using the same locale (simple scenario) and where application components are running on heterogeneous platforms and in different locales. You have to address issues relating to character sets, language identification, and cultural requirements when dealing with these issues. Certain standards bodies are currently trying to address such issues. These include the Unicode Consortium and W3C who have an internationalization working group dealing with such issues in the World Wide Web.

Storage and Interchange

File names should correspond to the translation retrieval mechanism used. For example, if you are relying on Java to select the resource bundle according to the locale, the class name and file name of the resource bundle must have the appropriate locale extension.

Application Programming Interfaces

Ensure you use the appropriate APIs for your platform when establishing the target locale or language. Most platforms have dedicated APIs for this. For more information, see the [Solaris example](#). Make sure that these APIs display appropriate default behavior when locale information is not available.

4.1 Translatable Product Components

4.1.2.1 Fixed Textual Objects

Description

Fixed textual objects are objects that should not be translated. Ideally, international developers should ensure that these objects are not internationalized in the first place. If this can be done, it substantially reduces the risk of problems later. However, developers often do not have control over the accessibility of fixed text objects by translators. Most translators are not programmers, and have difficulty distinguishing between programming logic, comments, and text information. This is often the case in files that contain both fixed textual objects and textual objects, for example, X resource files. Although this is not within the scope of internationalization, the provision of a special editing tool to hide fixed text objects from translators would prevent fixed text from being translated. Examples of fixed text objects include:

- **User or group names, passwords, hostnames, and shell and environmental variable names**

On Solaris and for the Internet, these objects are restricted to 7-bit ASCII for the moment.

- **UNIX commands and command line options**

`ls -l` for example is still `ls -l` in all locales.

- **Device names**

The translation of strings without concern for the actual use of these text items almost inevitably leads to problems. On a Microsoft Windows machine, for example, imagine the problems that would result from translating the word *device* in the device specification string `device=c:****.sys` in the `config.sys` file. The device driver in question would obviously not load due to the software not understanding the translated string. Likewise, on a UNIX machine, a string like `var/spool/mail` would not be translated; otherwise the mailer would fail to find the device used for spooling mail.

- **Some resources**

The following are examples of X resource fixed textual objects that should not be translated:

X color resource value: `*background: burlywood`

X font resource value: `*fontList: serif.r.14`

- **Other configuration file settings**

For example, in the http configuration file `httpd.conf`, the following text should not be translated:

```
AddLanguage    .french    fr
```

Command Line Interface

Options in command line interfaces are usually fixed textual objects, as in the previous `ls -l` example. This is true even though the particular option in question can be derived from an English word, for example, the `l` parameter in `ls -l` is an abbreviation of *long*.)

Character Interface

Fixed textual objects can be used in character interfaces. The `vi` editor is a good example of a character based interface. It provides a rich set of commands for text manipulation, most of which are derived from English words. For example, the `s` command can be used to substitute one string for another. This command would not work if the `s` were internationalized and subsequently translated to the first character of the local word for *substitute*.

Graphical Interface

Fixed textual objects may be used in graphical user interfaces (GUI). The problems associated with internationalizing and inadvertently translating fixed textual objects, as described previously, permeates through to the GUI, where control of the application often resides. A GUI implementation is a consumer of the operating system's services, that is, the provider; therefore, the GUI implementation is directly affected by any underlying internationalization problems inherent in the provider. On UNIX systems, for example, default device names are often used for printing and mail applications to indicate printer or mail spooler locations. Users often use these default settings as cues to the real locations. If these device names are translated, it is more difficult for users to guess the real locations.

Application Protocols

Fixed textual objects can be used in application protocols, such as HTTP for web browsing, SMTP for email, NNTP for broadcast communication, and LDAP for directory access. HTTP, for example, is the protocol used to transfer web pages between a server and a client. HTTP uses fixed text objects as part of its internationalization support, some of which are described in the following table.

Table 4-3. Use of Fixed Textual Objects in Application Protocols

Fixed Text Object	Description	Sample Client/Server Output
<code>charset</code> parameter	Specifies the character encoding of the document	<code>Content-Type: text/html; charset=iso-8859-1</code>
<code>Content-Language</code> response header	Indicates the languages in a page sent from the server to the client	<code>Content-Language: da</code>

Accept-Language	Restricts the set of natural languages that are preferred as a response to a request from the client	Accept-Language: da
-----------------	--	---------------------

It is important that the client/server output is not internationalized and subsequently translated, since the output consists of standard parameter names that are recognized by browsers worldwide, for example, `Content-Type`. The output can also consist of standard parameter values that might have been registered with specific standards bodies. HTTP is an example of where server and client are both providers and consumers of internationalization related information.

Storage and Interchange

Storage

Fixed textual objects can be used in certain file formats. These tend to be the richer file format types, such as HTML, XML, and SGML. In this case, the fixed textual objects are tags that play a key role in enabling applications to interpret the document. They should never be internationalized. However, authoring or editing tools that create or modify files with rich file formats do require internationalization. It is important, for example, that editing tools protect document tags from translation. We can use the feature of language tagging in HTML and XML to illustrate the importance of not internationalizing and localizing fixed text objects in storage.

Language codes can and should be used to indicate the language of text in HTML and XML documents. For [HTML 4.0](#), language codes are specified with the `lang` attribute. For XML, language codes are specified in the `xml:lang` attribute. In both cases, language information is inherited along the document hierarchy; that is, inner attributes override outer attributes. The language has to be stated only once if the whole document is in one language. HTML and XML rely on [RFC 1766](#) to define language codes. RFC 1766 is in turn based on ISO-639 two-letter language codes, and on [ISO-3166](#) two-letter country codes. Examples include:

`en`, `en-US`

Changing these tags without changing the text they describe can result in undefined behavior.

Interchange

Interchange protocols can contain fixed text objects. An example of this, is the property in inter-client communication in X. Properties are set directly by one application and read by another. They enable you to associate arbitrary information with windows, usually to make that data available to the window manager or other applications. A pre-defined property is uniquely identified by an integer ID called an *atom*; however, for non-predefined properties an ASCII string is also used. This string is used most when applications need to find out the

atom for a property for the first time using `XinternAtom()`. This returns the atom for the property, which is used to access the property from then on. If this property string were internationalized and localized, it might cause unpredictable behavior.

Application Programming Interfaces (APIs)

Function names and their arguments are usually fixed text objects. They should never be internationalized. Compilers should flag this for you.

Requirements for Compliance

1. Determine whether the text should be translated--use a common sense approach to do this.
2. If the text should not be translated, do not internationalize the text.
3. If the text appears in resource or configuration files, you can do either of the following:
 - Provide a tool that prevents any fixed text from being changed.
 - Provide detailed comments in the resource or configuration files indicating whether text should be translated.

Command line interface

See "Requirements for Compliance."

Character interface

See "Requirements for Compliance."

Graphical interface

See "Requirements for Compliance."

Application protocols

See "Requirements for Compliance."

Storage and interchange

See "Requirements for Compliance."

Application programming interfaces

See "Requirements for Compliance."

Character Interface

Messages are used in character interface components. These include menu items, labels, field names, headers, help systems, and error and status messages.

Graphical Interface

Messages appear in GUIs as menu items, labels, buttons, dialog boxes, help systems, and error and status messages.

One key point to remember about messages in graphical or character based user interface components, is that when they are translated they often affect the layout of the interface. For example, problems such as misalignment of interface components or truncation of text can occur. The reason for this, is that the majority of applications are first developed in English speaking countries and then translated into other languages. The translations are usually longer than the original English text. If the application is unable to deal with the extra space required by the translated text, the user interface becomes unusable. The following table provides recommended allowances for expansion based on text length in English.¹

NOTE: The table refers to the number of characters in a message. The expansion required assumes two bytes per character for multi-byte languages, for example, Japanese. On the screen, Asian languages can expand vertically, in addition to requiring more byte storage.

Table 4-4. Recommended Allowances for Text Expansion in Messages

Length of English Text	Expansion Required ¹
Up to 10 characters	101 - 200 percent
11 - 20 characters	81 - 100 percent
21 - 30 characters	61 - 80 percent
31-50 characters	31 - 40 percent
50-70 characters	31 - 40 percent
Over 70 characters	30 percent

1. Uren, Emmanuel, Robert Howard, and Tiziana Perinott, *Software Internationalization and Localization: An Introduction*, Van Nostrand Reinhold, June 1993.

Application Protocols

Messages can be used in interprocess communication. An example of this, is the passing of a window title and icon name from a client application to a window manager on X using the Interclient Communications Conventions (ICCC). Both the window title and icon name are messages and reside in an X resource file. Since ICCV uses compound text for interchange, the window title and icon title messages can be translated into localized text. In this example, there is an interesting provider/consumer relationship. The window manager, that is, the consumer, is enhanced so that it always converts the client title and icon name passed from the client's encoding to the encoding of the current locale. In this case, the client is the provider. An XmString is created using the XmFONTLIST_DEFAULT_TAG identifier. Thus, the client title and icon name are always drawn using the window manager's default

font list entry.

Storage and Interchange

Storage

Messages can be stored in different file formats and file encodings. They can also be shipped in either of the following formats:

- **Human-readable format** - For example, X resource files or proprietary file formats
- **Binary format** - Created from human-readable files, for example, Java resource bundles or XPG4 message catalogs

The file formats are many and varied, but all fall into the following categories:

- Plain text file with little or no formatting information, sometimes known as a "flat text" file.
- Plain text file with formatting information, for example, HTML and XML.
- Binary file, which is separate from the main executable, and usually language or product dependent, for example, Java resource bundle, XPG4 message catalog, and Microsoft Windows resource files.
- Executable with messages embedded.

Messages must be stored external to the main source code to allow for localisation. The storage format is product, language, or technology dependent. Tools or mechanisms for extracting text automatically from source code and storing it in a particular format are available on various platforms. If a suitable tool cannot be found, scripts or simple programs can be written to handle the extraction.

Interchange

It is important that messages or textual objects do not get corrupted during interchange between one application or process and another. An example which highlights internationalization issues with messages in interchange, is the issue of charset conversion in mail systems. When a provider (sending mail system) sends email to a consumer (receiving mail system), information relating to charset is passed using standard MIME headers (see [RFC 2045](#)). The receiving mail system uses charset conversion routines to convert the text in the message body to a charset appropriate for its system or locale. Thus, when developing mail systems, you must ensure the integrity of your mail content--text, in this case--by first tagging all outgoing email appropriately and converting all incoming mails to the appropriate target charset.

Application Programming Interfaces (APIs)

Application programming interfaces can be used to access localized messages, that is, translated versions of original text. These messages need to be accessed by applications at runtime to ensure that the correct locale-specific messages are used. Special APIs must be used to retrieve this text. For example, there are two distinct APIs for message handling on Solaris: the `catgets()` family, which is used with `.msg` files, and the `gettext()` family used with `.po` files. In Java, resource bundles can be used. These are basically Java classes; that is `.java` files. The API `getString()` is used to retrieve localized text from

the resource bundles.

NOTE: In this case, the provider is the operating system, and the consumer is the software with the messages.

Requirements for Compliance

Command Line Interface

Providers must accommodate messages in all supported charsets, and be able to supply a locale-specific message.

Consumers must understand and specify the current locale and externalize the command text.

Character Interface

Providers must supply a message externalization and retrieval mechanism and accommodate text in the supported charsets. Provider tools should allow for dynamic or utility based resizing of components.

Consumers must externalize translatable messages and accommodate layout changes as a result of text expansion, using available provider tools. They must retrieve messages according to locale.

Graphical Interface

The requirements that apply to [character interfaces](#) also apply here for both providers and consumers. The following requirements also apply:

- Providers must enable text and graphical elements to be separated.
- Consumers must use provider functions to ensure that message text is stored separately from graphics.

Application Protocols

Providers must accommodate messages in all supported charsets and allow for descriptive charset and locale information.

Consumers must supply relevant charset and locale information along with the message.

Storage and Interchange

Providers must supply a storage mechanism that allows messages to be stored separately from code. Interchange must be available for data in all supported charsets.

Consumers must store messages separately from code and provide relevant charset and locale information for proper interchange.

Application Programming Interfaces

Providers must ensure that APIs are available for locale-specific message storage and retrieval.

Consumers must use these APIs to access localized text.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.1 Translatable Product Components

4.1.2.3 Help Systems and Documentation

Description

Internationalization of textual objects also applies to help systems and hardcopy and software based documentation. Help systems are often separate applications that use different technologies for handling internationalization of textual objects in graphical or character based interfaces. However, the internationalization principles and issues that relate to software components also apply to help systems. The principles outlined for software components also apply to software based documentation. However, hardcopy and software based documentation have a unique set of linguistic requirements that should be approached differently from software internationalization. These linguistic requirements are discussed in detail in the ["Recommendations for Documentation"](#) section.

One key point highlighted in this section is the need to keep software, help, and documentation "synchronized". Synchronized here means using the same source for software, help, and documentation terminology where possible. This simplifies the localization process.

Command Line Interface

Command line interfaces often have basic help information, which you can usually access using a `-h` option. The internationalization issues here are the same as those outlined in the Command Line Interface subsections of sections [4.1.2.1](#) and [4.1.2.2](#).

Man pages, which provide help on UNIX commands, are another command line interface.

Character Interface

Help systems are often character based with character user interfaces (UIs). The issues discussed in the Character Interface subsections of sections [4.1.2.1](#) and [4.1.2.2](#) with regard to textual objects also apply here for online help and documentation systems. With any interface, it is always advisable to use the same technology for the online help and UI. As well as making synchronization of textual objects easy, it also means consistent internationalization technology across the software architecture.

Graphical Interface

Some help systems have a graphical user interface (GUI) provided by a special online viewer or browser. The issues relating to textual objects, as discussed in the Graphical Interface subsections of sections [4.1.2.1](#) and [4.1.2.2](#), also apply to online help and documentation systems. The comments relating to synchronization in the Character Interface subsection also apply here.

Application Protocols

Not applicable.

Storage and Interchange

Help and documentation data can be stored in different formats depending on the viewer or browser technology. They can also be interchanged between client and server and presented to users in user or system defined formats.

Application Programming Interfaces

Not applicable.

Requirements for Compliance

These requirements apply to the following interfaces:

- Command line interface
- Character interface
- Graphical interface
- Storage and interchange

There are no requirements for application protocols and application programming interfaces (APIs).

To ensure proper internationalization for help and documentation systems, you must ensure that the following provide full internationalization support:

- Markup language
- Authoring tools
- Illustration tools
- Online viewers
- Web browsers

To ensure synchronization of information, use the same technology to both author and render help and documentation systems.

Recommendations for Documentation

Text

1. Develop a standard glossary, and translate consistently across documentation sets.
2. Take into consideration relevant external glossaries, such as industry-standard terminology. If dealing with an after-market product, consistency should be maintained with the main product.
3. Establish standard linguistic style guidelines.
4. Maintain consistency between the documentation and the software's UI.
5. Use clear, concrete language.

6. Avoid slang and idioms.
7. Define acronyms and abbreviations.
8. Avoid cultural references, such as gender-specific roles, humor, ethnic or historical references.
9. When choosing to include cultural references, encapsulate such content into well-defined locations.
10. Customize formats to fit the target country, for example date and time formats and currency formats.
11. Confirm that your localization partner has a leveraging strategy, including any translation memory implementations, for reuse of translations and graphics that have already been localized. This strategy should accommodate both new product releases as well as updates during the localization process.

Layout

1. Establish standard layout guidelines, clearly defining the use of design styles
2. Consider the development of a universal template to reduce layout tasks and time during localization.
3. Use standard fonts that are well supported by most output devices and are easily available. If special fonts are used, it is critical to communicate this to the localization team up-front, prior to localization.
4. Save call-outs, as well as text within tables, as text rather than as graphical elements
5. Leave sufficient white space to accommodate text expansion during translation.
6. Anticipate font type and size modifications for some character sets.
7. Anticipate vertical expansion of the text for some character sets, for example, Traditional Chinese.
8. Avoid formatting characters and paragraphs manually. Instead, define an appropriate style to automate the task and ensure consistency.
9. Automate header and footer text, such as chapter titles and page numbers. This information can change during repagination of the localized documentation.

Graphics

1. Use standard, easily available applications to create graphics in target languages.
2. Avoid using text in icons and graphics, if possible.
3. If text is included, try using callouts or captions with the text component in the DTP application rather than in the graphics file. Otherwise, if the text is within the graphic itself, save the text as editable text, not as pixels.
4. Assume that callouts and text within graphics will increase in length when translated. Leave sufficient white space to accommodate this text expansion.
5. Link graphics rather than embed them whenever possible.
6. For screen captures, note the computer configuration and software settings, such as color depth and screen resolution. Give this information to the localization team.
7. For screen captures requiring complex setup and navigation, identify these and supply

regeneration instructions to the localization team.

8. Avoid using culturally-specific icons and graphics.

9. Avoid using representations of people and animals in icons and graphics.

Courtesy of Rubric, *Tips for Creating World-ready Documentation*,
[<http://www.rubric.com>].

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.1 Translatable Product Components

4.1.3 Non-Textual Objects

Description

This section provides guidelines for internationalizing non-textual objects. It contains the following subsections:

- [4.1.3.1 Icons, Images and Colors](#) - Non-textual elements, such as icons, need to be optimized for generic, universal use. This section also describes cultural considerations for using images and choosing color schemes.
- [4.1.3.2 Graphical User Interface \(GUI\) Objects](#) - This section describes cultural considerations for laying out GUIs.
- [4.1.3.3 Sounds](#) - This section describes cultural considerations for using sounds.
- [4.1.3.4 Other Non-textual Objects](#) - This section describes other non-textual aspects of internationalization, such as video, accessibility issues, and naming conventions.

Recommendations for Compliance

Providers should isolate locale-specific data. Resource bundles should store anything that can vary across localized versions. This includes locale-specific text, fonts, mnemonics, color schemes, icons, graphics, and sounds. It is best to keep non-textual elements in separate resource files or separate sections from textual elements.

Perform usability tests with users from the target locales. This identifies graphics that are offensive or difficult to understand and can also help to verify the accuracy of the translated text. Usability studies can achieve great results. If users in one locale have trouble understanding something in the product, often users in another locale have similar difficulties. The scheduling of these usability tests should accompany rather than follow the product development cycle so that there is time for enhancements before the product ships.

Engage an interface designer who is trained in international user interfaces to evaluate the graphical elements and layout of the interface.

4.1 Translatable Product Components

4.1.3.1 Icons, Images, and Colors

Description

Non-textual elements, such as icons, need to be optimized for generic universal use. This section also describes cultural considerations for using images and choosing color schemes.

Command Line Interface

Not applicable.

Character Interface

Not applicable.

Graphical Interface

Graphical user interfaces (GUIs) use icons, images, and color in a variety of places, including the toolbar buttons, the splash screen, progress animations, and the product icon.

Graphics are important because when properly used they can convey a large amount of information very quickly. In some countries, such as China, pictorial representations are especially common and familiar to people due to their prominence in the language and culture.

Although the most colorful parts of graphical interfaces are often buttons or icons, color can also be used in more functional ways in the interface. For instance, color can be applied to text to indicate status. A disk name in green can mean that the disk is working smoothly, while a disk name in red can represent a disk failure. However, the connotations of red and green are not the same in every country.

Application Protocols

Some application protocols accommodate language or locale based data. A protocol string might include an element that contains language information or a locale. For example, in HTTP, a client can send an Accept-Language header, which tells the server the client's preferred language version of the requested file. The file could be a graphic, which may change according to the language.

Storage and Interchange

Like messages, images and icons are often stored by locale, separate from code. They are not usually part of another resource file, rather they are stored individually. Image formats themselves are not locale-sensitive.

Application Programming Interfaces (APIs)

Not applicable.

Requirements for Compliance

Command Line Interface

No requirement.

Character Interface

No requirement.

Graphical Interface

Graphics should be appropriate for all localized versions of the product. Graphics should not have to be localized for each version. Successful generic graphics are ones that leverage existing international symbols, such as those used in airports.

Icons should make use of an icon repository. When icons are reused across various products, users find them easy to recognize.

A graphic designer, preferably someone with experience in creating graphics for products outside the United States, should review the product's graphical elements.

Graphics should be tested by showing them to users in the target locales. A low-cost way to do this is to email the proposed icons to salespeople in different locales for feedback.

Tooltips should be used to describe icons, buttons, and other graphics. Tooltips not only help people understand the images, but they are also required for people with visual impairments who use a text-reader to decipher images. Providers should store tooltips as localizable resources.

Just as tooltips help users decipher graphical elements, textual alternatives should also be employed when color conveys information. This approach accommodates users who do not understand the intended significance of a particular color scheme, as well as users who are colorblind. Providers should store these textual alternatives as localizable resources.

The following types of images should be avoided:

- **Images that contain text** - If an image contains English text, it needs to be redesigned for each locale.
- **Images that contain numbers** - Numbers can have different connotations in different locales. Just as the number 13 has an unlucky connotation in the United States, the number 4 connotes death in both Japan and Hong Kong.
- **Images that contain hand gestures** - A gesture that is appropriate or meaningful in one locale can be offensive or meaningless in another locale.
- **Images that present a play on words** - Puns do not translate well.
- **Images of animals** - Just as the image of a dog to represent food would be unsettling to most people in the United States, the image of a cow in the same context can offend people in India.

- **Images of people or faces** - Depictions of certain facial expressions, nontraditional gender dynamics, and uncovered skin can be offensive to users in some locales.

Colors should not be referred to by name in the interface, since the words that people from different cultures use to describe the same color can be different. A color that looks green to a user in the United States could look blue to a Japanese person, for instance. Also, people in the United States call the middle color of a traffic light "yellow", while people in the UK call it "amber".

Application Protocols

Providers should supply a protocol that allows for locale specification with a graphic element, where relevant.

Consumers must give locale information when required in the protocol.

Storage and Interchange

Providers must supply locale-specific storage for graphic elements.

Consumers must store localizable graphic elements by locale.

Application Programming Interfaces (APIs)

No requirement.

4.1 Translatable Product Components

4.1.3.2 Graphical User Interface (GUI) Objects

Description

This section describes cultural considerations for laying out GUIs.

Command Line Interface

Not applicable.

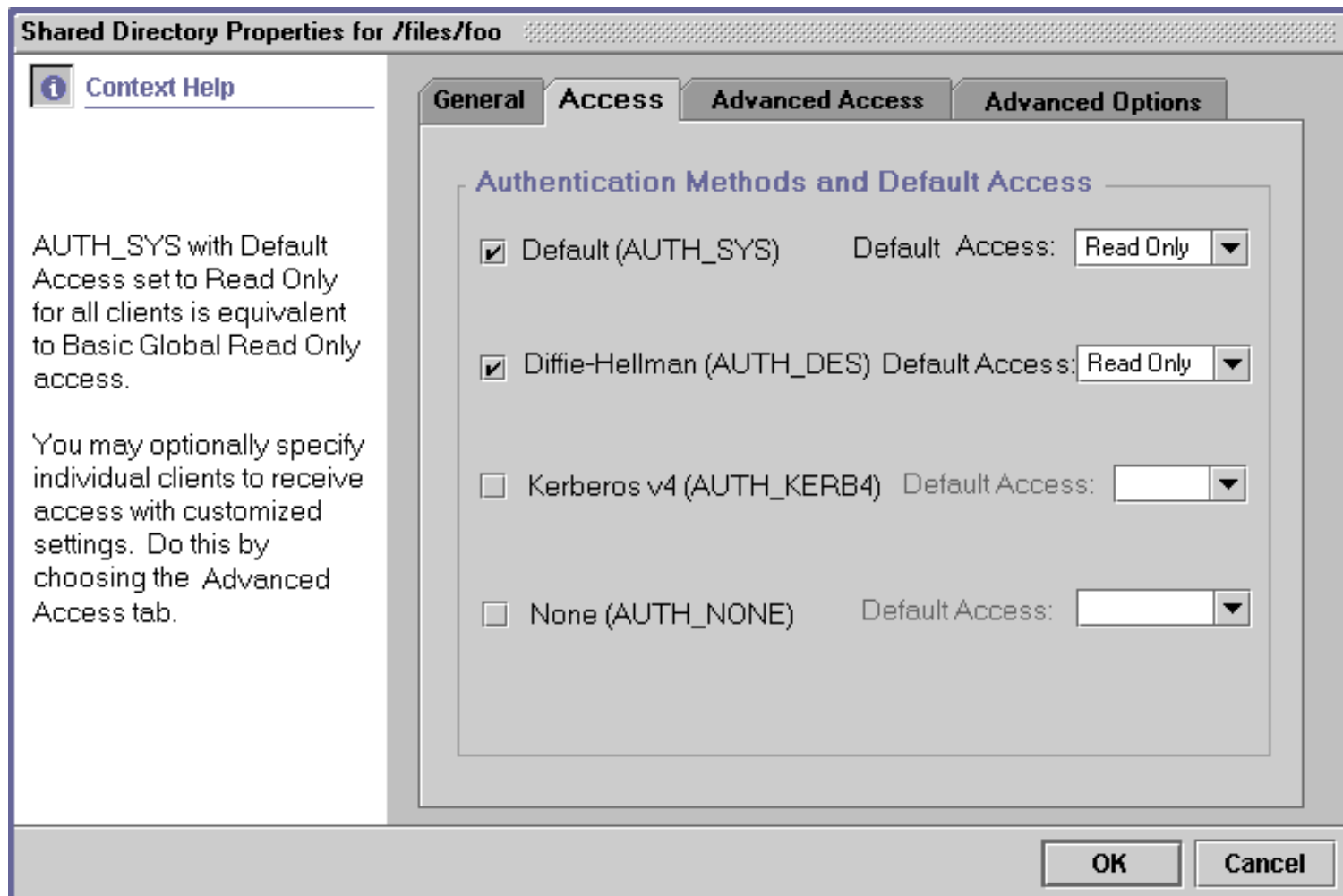
Character Interface

Not applicable.

Graphical Interface

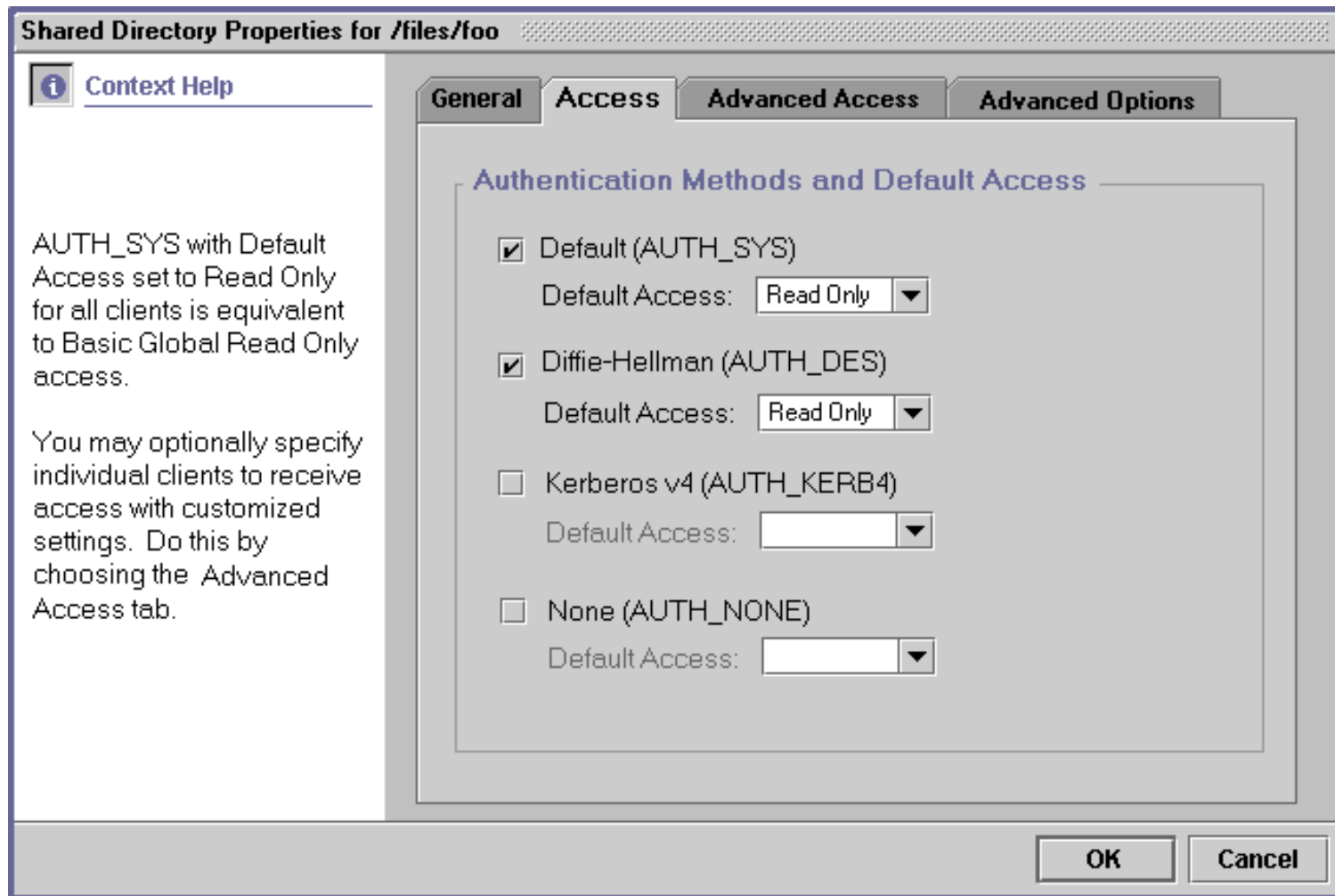
When dialog boxes are translated from English into other languages, the length of the components that contain text can increase by more than 30%. The components that are affected by translation include buttons, menus, tables, labels, and text fields. If the window is already wide, expansion can cause it to exceed the width of the screen. The height of textual components can also increase when they are translated into Asian languages, since Asian characters can be unreadable at a Latin height.

In the following example, the components of a dialog box are laid out side-by-side. The dialog box will become much wider after translation.



AdminSuite 3.0 Dialog Box - Courtesy of Karen Stanley

The dialog box is redesigned by stacking the components vertically. Now, there is space to the right of the components for expanded text.

*AdminSuite 3.0 Dialog Box - Courtesy of Karen Stanley*

Word order changes from one language to another. A GUI that embeds a component in a text string must be laid out again when translated into French. In French, most adjectives come after the noun they describe instead of before it. In the following example, the French translation has an incorrect word order.

*Graphic courtesy of the Java Look and Feel Design Guidelines*

To anticipate variations in word order, a layout like the following should be used in the original version of the product.

*Graphic courtesy of the Java Look and Feel Design Guidelines*

Now the word order is correct in both French and English.

Application Protocols

Not applicable.

Storage and Interchange

Not applicable.

Application Programming Interfaces (APIs)

Not applicable.

Requirements for Compliance

Command Line Interface

No requirement.

Character Interface

No requirement.

Graphical Interface

Dialog boxes should be designed to resize gracefully when text expands, so the localized user interface is attractive and readable. GUI objects should be cushioned amidst ample horizontal and vertical space. This way, they can expand into the blank space when translated; hence reducing the possibility of the entire dialog box growing.

If the total length of a row of GUI objects on a dialog box is longer than 60% of the width of the dialog box, or if the total height of a column of GUI objects is taller than 80% of the dialog box, the layout probably needs to change.

When layout managers are available for the platform, providers should use them in laying out dialog boxes. This way, when the components are translated, the whole window can expand automatically if necessary.

Layouts should be tested with a text expansion utility. The utility should test horizontal expansion by appending a certain number of extra characters to textual components. If a resulting expanded text string is too long for the window, the layout needs to be fixed. The utility should also test vertical expansion by increasing the height of all textual components by a specific amount. If the results make the dialog box unreasonably tall, the layout needs to be fixed.

Providers should ensure that the location of GUI objects on a window and the tab traversal order are pluggable. If the application is translated into a language with a right-to-left reading order, the layout of the GUI objects must be reversed. For instance, vertical scrollbars must move to the left side of the window and labels must be placed to the right of their associated GUI objects. On some systems, this happens automatically; this automatic shift should be verified. Also, providers should verify that the tab traversal order makes sense in a right-to-left locale.

If the application is translated for a locale with a right-to-left reading order, its layout should be examined by literally looking at the user interface in the mirror.

GUI objects should not be embedded within a text string, since the order of words can change from one language to another.

Application Protocols

No requirement.

Storage and Interchange

Graphical User Interface (GUI) Objects

No requirement.

Application Programming Interfaces (APIs)

No requirement.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.1 Translatable Product Components

4.1.3.3 Sounds

Description

This section describes cultural considerations for using sounds.

Command Line Interface

Not applicable.

Character Interface

Not applicable.

Graphical Interface

Sounds can be used in a product to indicate status or to signal an error. The meaning of sounds can differ from one country to the next. A ringing telephone and a siren sound differently in different countries. Also, people can react to sound in different ways. While it is common to hear computers bleeping error messages in a workplace in the United States, the audibility of mistakes is more embarrassing in a country such as Japan, where people work closely in small groups, often with a supervisor within earshot. It is important to remember that hearing-impaired users require alternatives to sounds.

Application Protocols

While images can be made generic, it is difficult to make generic sounds. Protocols that involve sound requests can have language or locale parameters as part of the string.

Storage and Interchange

Like messages, sounds can be stored by locale, separate from code. They are not usually part of another resource file, rather they are stored individually. Sound formats themselves are not locale-sensitive.

Application Programming Interfaces (APIs)

Similar to application protocols, APIs can provide locale-specific sound data.

Requirements for Compliance

Command Line Interface

No requirement.

Character Interface

No requirement.

Graphical Interface

Providers should treat sounds as localizable resources. To facilitate localization, sounds should be stored in external files. If a product contains sound, it must offer an easy way for users to turn the sound off. Also, if a product uses sound to convey information to the user, it should always provide an alternative textual format. Providing alternatives to sound accommodates users in a variety of locales, as well as people with hearing impairments.

Application Protocols

Providers should supply a protocol that allows for locale specification with a sound element, where relevant.

Consumers must give locale information when required in the protocol.

Storage and Interchange

Providers must supply locale-specific storage for sound elements.

Consumers must store localizable sound elements by locale.

Application Programming Interfaces

Providers should supply a protocol that allows for locale specification with a sound element, where relevant.

Consumers must give locale information when required in the protocol.

4.1 Translatable Product Components

4.1.3.4 Other Non-textual Objects

Description

This section describes other non-textual aspects of internationalization, such as video, accessibility issues, and naming conventions.

Command Line Interface

Not applicable.

Character Interface

Not applicable.

Graphical Interface

Some other non-textual aspects of internationalization include:

- **Video** - Video is often used for product demonstrations and marketing purposes. Since it is expensive to localize video, the voiceover is often the only part that changes for each geographic market. Different cultures have different ideas of what is appropriate as far as movement, degree of eye contact, and clothing are concerned. The non-textual aspects of internationalization that apply to GUIs are present and even intensified in video format. Also, video formats can differ from one country to another.
- **Accessibility** - It is required by United States law that all products, including software applications, accommodate users with disabilities. Other countries might not have similar laws; however, it is helpful for products to be accessible to people with disabilities outside the United States as well. Accessible software applications accommodate users who are blind or colorblind, users with hearing impairments, and users with motor skills impairments that prevent them from using the mouse.
- **Naming conventions** - The name given to a feature in the original version of a product can affect how that feature is translated and hence connoted to users in different locales. Naming components after American concepts like "stickies" can make the components difficult to understand outside the United States. Also, users from cultures that value dignity and formality appreciate accurate, functional names more than informal or cute ones.

Application Protocols

Not applicable.

Storage and Interchange

Not applicable.

Application Programming Interfaces

Not applicable.

Requirements for Compliance

Command Line Interface

Not applicable.

Character Interface

Not applicable.

Graphical Interface

The graphical interface requirements for compliance can be categorized as follows:

- **Video** - The colors, graphics, and sounds used in video require the same considerations that they do in graphical user interfaces (GUIs). A marketing professional with experience marketing products internationally should be engaged to review the video before it is released outside of the United States.
- **Accessibility** - Keyboard equivalents should exist for all mouse actions. This is imperative for people who do not use a mouse, as well as for blind people who use a screen reader to decipher the interface. Anywhere sound or color is used to convey meaning in the interface, there should be a textual equivalent. This accommodates users with hearing or vision impairments.
- **Naming Conventions** - When deciding between a functional name and a more informal name for a feature or component, the functional name should always be chosen. This results in more meaningful and accurate translations.

Application Protocols

Not applicable.

Storage and Interchange

Not applicable.

Application Programming Interfaces (APIs)

Not applicable.

4.2 Cultural Formatting and Processing

4.2.1 Culture Negotiations, Defaults, and Selection

Description

Cultural conventions affect much of the data people see every day. Numeric formats, such as date or prices, addresses, even page layout can change from culture to culture. In order for an application to provide information in the format a user expects, appropriate cultural information must be accessible from somewhere. The determination of culture can be done by asking the user to choose, by a configuration setting in the application, or by detection of system settings. Settings can remain for a single object, a session, or until they are changed manually in a properties or configuration file.

In the language of software development, a culture is called a locale. Locale formats can vary from platform to platform, though there are some guidelines and de facto standards. Applications manage the translation from a software locale designation on one platform to that of another, to something which is understood by the user.

In this case, providers must supply a locale retrieval mechanism or environment setting. In addition to using provider locale determination, consumers might need to provide a method for the user to communicate a specific locale.

For more information, see ["Translation Negotiations, Defaults, and Selection."](#)

Command Line Interface

Command line functions can determine locale from the system, the environment, or as part of the command argument. Some command line functions use all three methods in a hierarchy of availability: If there is no locale argument, then check the environment. If there is no environment locale setting, then check the system. Other commands take an argument or fall back to a default locale.

Character Interface

Character interfaces lend themselves to two types of locale determination: The locale can be read from the system or configured as a user setting. Some interfaces have a fallback locale, which in some cases is configurable, for example, the command line interface.

Graphical Interface

Similar to character interfaces, the graphical interface usually provides a user-settable locale with some sort of fallback mechanism to a system or default locale.

Application Protocols

Protocols do not have the ability to query settings and so include explicit locale information as parameters.

Storage and Interchange

Storage and interchange, like protocols, cannot use system settings. One machine can store data in several locales, so the data must be stored with explicit locale information. Some data can be stored in a base format with locale-specific formatting determined by a user interface level. An example of this is dates; they are often stored in [coordinated universal time \(UTC\)](#) values and a specified base format. The user interface code then reformats the date into the locale of the current user who requests it.

Application Programming Interfaces (APIs)

APIs can require explicit locale information, use system or environment settings, or do both in hierarchical sequence.

Requirements for Compliance

Providers must supply structures or mechanisms for locale retrieval either from the environment or from system settings.

Consumers must use provider locale information, but should also provide as much flexibility to the user as possible. Product usability improves if you enable users to set their preferred locale.

Command Line Interface

Providers must supply an environment structure.

Consumers should allow for a user-specified locale and have a fallback to environment, system, and default settings.

Character Interface

Consumers should query the user and have a fallback or default locale. At a minimum, consumers must have a locale configuration setting.

Graphical Interface

Same as "Character Interface."

Application Protocols

Providers must allow for locale parameters in the protocol, where appropriate.

Consumers must supply locale information in the protocol.

Storage and Interchange

Providers must allow some method for explicitly stating the locale of related data or force the data into a published neutral format.

Consumers must supply locale information with data for storage or interchange and read the locale when retrieving the data. They must format the data according to locale, where appropriate.

Application Programming Interfaces

Providers must supply parameters for explicit locale inclusion, where appropriate. They must publish locale handling, such as fallback to an environment or system locale. It is strongly recommended that the locale parameter scope be limited to a thread or object.

Consumers must supply locale parameters, where possible, and handle fallback situations as published by providers.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.2 Cultural Formatting and Processing

4.2.2.1 Time, Date, and Calendar

Description

The conventions to express time, date, and calendar data tend to vary by country and language: For example, the date November 22, 1999 is displayed in the mm/dd/yyyy format in the U.S. as 11/22/1999, but in the U.K., it is displayed in dd/mm/yyyy format as 22/11/1999. The ISO 8601 standard describes time formats. This standard has been widely adopted in Europe.

The conventions for expressing time also differ, with varying formatting schemes using AM/PM, 24-hour clock, and designations for Daylight Saving Time.

While many countries use the Gregorian calendar, other calendars, such as Buddhist (Thailand) or Lunar (Middle East and Asia) are in use in other parts of the world. Even conventions such as the first working day of the week can vary between countries. While most Western countries consider Monday as the first working day, in the Middle East, Sunday is considered to be the first working day. This can be important when displaying calendar appointment data.

Application software must be able to format and display the time, date, and calendar data according to local cultural conventions. The assumptions about these formats must not be embedded in the application software.

Software that provides interfaces for formatting this data according to cultural conventions or for modifying these cultural conventions is considered a provider. Software that takes advantage of the provider functions is a consumer.

Command Line Interface

At the command line, users entering time, date, or calendar information want to use familiar formats. Commands that take this data as parameters should take the current locale into consideration when parsing. Output data can be in an international standard format or locale-specific format, depending on the nature of the command. For example, if a command is intended for use in scripts or programs with the output data then fed to other programmatic functions, it makes more sense to use a single standard format. If the command is intended for human use, it is logical for the data to be in a locale-specific format.

Character Interface

Many character interfaces break up time, date, and calendar data into their component segments. For example, there might be three fields: one for month, day, and year. These fields are often rearranged to suit various locale needs. In the U.S., the name of the month comes first, followed by the day, a comma, and then the year. For Germany, the day comes first, followed by a period, the name of the month, and then the year.

Graphical Interface

Similar to the character interface, the graphical interface often has time, date, and calendar data broken up into components. You can arrange these components in either of the following ways:

- Generically for international use, for example:

Time	UTC (GMT)
Hours	02
Minutes	45
Seconds	00

- According to the current locale, for example:

Time: :

Application Protocols

Time and date information is commonly passed in protocols. Many transactions passed through protocols require a date and time stamp; for example, financial transactions, email messages, and status messages. In most cases, the information is passed using the coordinated universal time (UTC), sometimes referred to as Greenwich Mean Time (GMT). Local time zone information is then managed at the client level for display and input. The time is converted to UTC before using the protocol. In some situations, the protocol takes the time zone information as either part of the time format, as an offset to UTC, for example, or as a separate parameter.

Storage and Interchange

Date and time are frequently stored in association with other data. They can be stored in local format with the associated locale and time zone for context or they can be stored in a universal format, to be converted to local format at the point of display. Calendar information is often stored on its own, for reference by functions that need to format date and time data into local calendar format.

Application Programming Interfaces (APIs)

In APIs, date, time, and calendar information can be managed in many different ways. For example, the API might:

- Break data up into components for separate processing, for example, the month portion of a date.
- Expect a full date-time stamp in a specific format for processing, with a separate

parameter to indicate the preferred calendar.

- Assume a format based on the current locale.

Requirements for Compliance

Providers must use functions that supply:

- Locale-specific formatting for date and time.
- Parsers for dates and times entered in locale-specific format.
- Localized display names for days of the week and months of the year in long form and in short form. For example, *Sunday*, *Monday*, and *Tuesday* are long form and *Sun*, *Mon*, and *Tue* are short form.
- First day of the week.
- Locale-specific calendar with time arithmetic capabilities, such as values for next day, *n* days later and the *n*th day of the year.

Optionally, the provider can supply functions that modify cultural conventions under programmatic control.

The following requirements apply to consumers:

- Make no assumptions about the display format of date and time. The consumer must use locale-specific formatters.
- Do not directly parse date and time formatted strings. The consumer must use the functions provided to parse the date and time text data entered in locale-specific formats.
- Use the locale-specific functions provided to manipulate calendar data or to do time arithmetic.
- Do not hard code messages that contain any format information for date and time, for example: `Enter date (mm/dd/yy)`.

Command Line Interface

Command line interfaces must consider their audience and either make provisions for entering time, date, and calendar information in a locale-specific format along with a possibility for locale information, or use an international or universal format. The required format must be documented.

Character Interface

Character interfaces must allow for rearrangement of components on the screen to enable different locale formats for displaying date and time. Local calendars must be used whenever end users need to enter dates and times.

Graphical Interface

Same as "Character Interface."

Application Protocols

Date and time should be exchanged in standard forms, leaving the formatting to the display level application. Calendar information should be exchanged as a separate parameter, where appropriate to the protocol.

Storage and Interchange

Date and time objects must be stored and interchanged in either a standardized form, or in a local form in conjunction with locale data.

Application Programming Interfaces

See ["Requirements for Compliance."](#)

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.2 Cultural Formatting and Processing

4.2.2.2 Numeric, Monetary, and Metric

Description

Numbers are commonly displayed using the Arabic numeral system; that is, digits 0-9. However, the convention to group the digits of a number varies from one culture to another. For example, in the U.S., a comma is used to separate digits into groups of three, for example, 10,000,000, whereas in France, the separator is a dot. In Japan, digits are separated into groups of four.

The convention for displaying monetary value varies from one country to another; that is, the currency symbol and the position of the currency symbol. For example, to represent 5 US dollars and 45 cents, you place the dollar symbol before the value--\$5.45. In some locales, the currency symbol can appear after the value or even between the integral and fractional designations; for example, Portuguese 5\$45 represents 5 escudo and 45 centavos.

Units of measurement can also vary from one locale to another. Standard metric units for length, volume, and weight include meters, liters, and grams. Standard imperial units include feet, ounces, and pounds. However, there are other region-specific measurements: Clothing sizes change from region to region; for example, a U.S. ladies' dress size 10 is a U.K. size 14 and a U.S. ladies' shoe size 8 is a European size 39. Paper sizes change, packaged quantities change, temperature scales change--anytime there is a measurement unit involved, it is likely to vary in different locales.

Command Line Interface

A provider receives a value from a consumer, and either receives the locale as well or retrieves it from elsewhere in the application or system. The provider then parses the data, supplying an internal format for processing or a locale-specific format for output. In some cases, the actual format is provided by the consumer module. Most numeric formats can appear in a command line, though in some cases, unit symbols are not available.

Character Interface

A character interface takes positioning into account, as well as the basic formatting necessary for command line. For example, the decimal position for monetary units might be separate from the numeric portion for the display. For metrics, the unit names can precede or follow the value, and so the interface is flexible enough to accommodate field swapping.

Graphical Interface

In a graphical interface, flexibility for positioning and input is built in, similar to that of a character interface. Some graphics might be used to represent units, although this costs more to localize. As in command line, a provider takes in locale and value information and parses the data accordingly. A consumer supplies the provider function with the necessary data for

processing.

Application Protocols

An application protocol can include fields for the value, as well as for the unit, or specify the unit so that the value is converted before it is passed using the protocol.

Storage and Interchange

Values might be stored in an internal format using a single unit for all stored values along with a conversion factor. Values can also be stored in an internal format along with their corresponding units. For example, dimension measurements in an application can be stored in meters, regardless of what units were originally entered. A set of conversion tables are then tied to the conversion process. However, product prices can be stored in their original currency and have a currency unit associated with each value.

Application Programming Interfaces (APIs)

APIs can include different combinations of parameters to allow proper numeric formatting. Sometimes the combination of value and locale is sufficient, sometimes a format mask or template is needed. Units might need to be specified as well. A flexible interface provides the most functionality, since many numeric formats are undefined.

Requirements for Compliance

Command Line Interface

Providers must supply functions that take locale into account along with a given value for input or output processing. If necessary for proper processing, the provider functions must also accept unit and format parameters.

Consumers must provide the necessary information to provider functions. They must not override provider functionality.

Character Interface

Providers must supply flexibility in field positioning, as well as numeric formatting functionality. See the [command line interface](#) requirements.

Consumers must lay out the interface to accommodate varying numeric formats and provide the necessary information to provider functions.

Graphical Interface

Providers must supply layout and numeric formatting flexibility, similar to that of the character interface.

Consumers must:

- Provide a flexible layout for varying numeric format.
- Supply provider functions with required information for proper numeric formatting.
- Keep graphic representations of units to a minimum.

Application Protocols

Providers must ensure that the protocol includes information necessary for proper numeric formatting.

Consumers must provide the necessary information in the protocol.

Storage and Interchange

Providers must supply a mechanism for the storage of values, which either:

- Converts them to a standard format using a standard method
- Retains enough information associated with the value to ensure that it can be properly formatted when retrieved

Consumers must store the values according to provider specifications.

Application Programming Interfaces

Providers must supply locale-specific functions to format numbers, currency, and metrics. The APIs must be flexible enough to allow for custom formats.

Consumers must supply the required information to the provider APIs.

4.2 Cultural Formatting and Processing

4.2.3.1 Ordered Lists (Collation)

Description

Collation refers to the comparing and ordering of data into sorted lists or categories. This is often a locale-specific function. The following table shows some locale-specific sorting sequences.¹

Table 4-5. Locale Specific Ordering

Spanish	German	C
a	a	a
à	à	après
âpre	âpre	azur
après	après	lase
âpreté	âpreté	lassen
azur	azur	laß
être	être	llama
lase	lase	luna
lassen	laß	à
laß	lassen	âpre
luna	llama	âpreté
llama	luna	être

1. O'Donnell, Sandra Martin, *Programming for the World: A Guide to Internationalization*, Prentice Hall, April 1994, p. 224.

Note that the Spanish "ll" sorts between *l* and *m*, so *llama* is sorted after *luna*. German does not have a special rule for *ll* so it sorts *llama* before *luna*. Likewise, *laß* is sorted before *lassen* in German, but it is the opposite for Spanish. German treats β as the two letters *ss*. Since Spanish does not include an β in its locale definition, it simply sorts it by its encoded value. The C locale sorts everything by its encoded value; hence the considerable difference from the other locales.

Sorting methods used throughout the world include:

- **Multilevel** - Involves secondary and tertiary sorting for tie breaks as well as primary sorting.
- **One-to-many** - One character sorts as if it were a multicharacter string. For example, the German β sorts as if it were *ss*.
- **Many-to-one** - A multicharacter string sorts as a single character. For example, the Spanish *ch* and *ll* strings are treated as single characters for sorting purposes. The *ch* is sorted between *c* and *d* and the *ll* between *l* and *m*.

Command Line Interface

The output generated by many command line interfaces is often influenced by locale-specific sorting rules. For example, the UNIX command `ls` lists files in a directory. The order in which these files are displayed on screen can differ depending on the ordering rules for the locale.

Character Interface

Sorting issues should not have a major impact on character based interfaces. It is important to remember, though, that character interface components cannot dynamically change as easily as graphical interface components. This has implications when dealing with ordered lists. If you have a single text component per list item, it is difficult to dynamically rearrange these according to different ordering rules without having a major impact on performance

Graphical Interface

Sorting is more of an issue for graphical interfaces. Components in graphical interfaces tend to be dynamic. This is because their layout is often determined by user actions. For example, window systems like Microsoft Windows or X Windows enable users to arrange file icons according to one of several different parameters, including size and name. The latter of these implies ordering. This means that the software rearranging the icons by name needs to understand the sorting rules for the locale in which it is running and position the icons accordingly. This is a simple example of ordering in graphical interfaces. A more complex example would involve using a graphical interface to enable users to manipulate text lists. A spreadsheet is a typical example of this. Each list item can be represented by a single graphical component, for example, button, text field, and label. If the user adds or removes items in the list, the interface must be dynamically rearranged to allow for any change in the ordered list due to the addition or deletion.

Application Protocols

Application protocols sometimes include collation data that can be used as part of an information request or provision. In a request, the data might be used to tell another machine to perform the operation, as in a client request to a server. In providing the information, collation data might simply describe the order.

Storage and Interchange

While information is not usually physically stored in a particular order, it is often indexed this way. Some storage systems need to know the language of the data in order to set up the index properly.

Application Programming Interfaces (APIs)

Collation APIs can sort based on many different criteria. Some use the numeric value of each encoded character, others use tables and algorithms. For results based on language, the

latter procedure is necessary. For example, here are some of the C sort functions:

Table 4-6. Sorting Functions

Function	Description
<code>strcmp()</code> and <code>strncmp()</code>	Sorts ASCII data - based on encoded value
<code>wscmp()</code>	Performs <code>wchar_t</code> sort - based on numeric values
<code>strcoll()</code> and <code>strxfrm()</code>	Collates data - char based
<code>wscoll()</code> and <code>wcsxfrm()</code>	Performs <code>wchar_t</code> sort

These functions use locale specific sorting rules. The `*coll()` functions are slower than than the `*cmp()` functions because the former use collation tables to determine order. To enhance performance, first use `strxfrm()` or `wcsxfrm()` which assign numeric values to characters using the current locale's sorting rules. `strcmp()` and `wscmp()` can then be used to do numeric comparisons. This is particularly useful for comparing the same data several times. Instead of including multiple calls to the slower table-driven `*coll()` functions, transform the characters once and then use `strcmp()` or `wscmp()` multiple times.

Example: Danish text sorts *æ*, *ø*, and *å* after *z*.

Suppose you need to sort some Danish text. Assume `strxfrm()` assigns numeric values as follows:

a
b
c
...
z
æ
ø
å

Given these assignments, suppose your program compares the strings *præst* and *prøve*. After running them through `strxfrm()`, they look like this:

<i>p</i>	<i>r</i>	<i>æ</i>	<i>s</i>	<i>t</i>
115	117	126	118	119
<i>p</i>	<i>r</i>	<i>ø</i>	<i>v</i>	<i>e</i>
115	117	127	121	104

When `strcmp()` looks at these transformed strings, it finds them equal up to the third letter, where it correctly determines that *æ* sorts before *ø*.²

2. O'Donnell, Sandra Martin, *Programming for the World: A Guide to Internationalization*, p. 219.

Requirements for Compliance

Command Line Interface

Providers must supply a mechanism for specifying the locale in command line sort functions and these functions must be locale-sensitive, where relevant.

Consumers that parse or manipulate output from shell commands must not make any assumption about the order of that output, and must either provide a locale to the sort function, or determine the locale used in some manner. For example, when using the `sort` command to order a set of names, the output might not be in English alphabetical order; the order is determined by the locale of the environment. Parsing the sorted data yields different results in different locales, possibly producing an error.

Character Interface

Consumers should always manipulate ordered lists within a single text field component.

Graphical Interface

Providers must supply a mechanism for specifying the locale in the creation of ordered list elements, and must sort them according to the specified locale.

Consumers must ensure that any graphical components that deal with ordered lists can be dynamically rearranged as a result of user or program action. They must display sorted data according to the user's locale where relevant.

Application Protocols

Providers must include locale information in a protocol sorting request.

Consumers must provide locale information according to the protocol when requesting sorted information, or read the locale when receiving sorted information.

Storage and Interchange

Providers may use locale information for indexing stored data.

Consumers must supply locale information to provider storage which requires it for indexing.

Application Programming Interfaces

Providers must allow locale parameters in sort APIs and must sort data according to the given locale .

Consumers must include a locale when calling a sort API.

4.2 Cultural Formatting and Processing

4.2.3.2 Personal Names, Honorifics, and Titles

Description

Names, titles and various forms of personal greetings can vary widely among cultures. In Japan, for example, given names are rarely used. The suffix *-san* is usually added to the family name, for example, *Tanaka-san*. In China, the family name comes first and is usually one syllable. It is followed by a one or two syllable given name. For example, in the case of *Teng Peinian*, *Teng* is the family name and *Peinian* is the given name. Or, in *Lai Pan Fu*, *Lai* is the family name and *Pan Fu* is the given name. However, when visiting and sending letters abroad, some westernized Chinese might reverse their names, Western-style, such as *Peinian Teng* or *Pan Fu Lai*. Some countries also use special characters for displaying foreign names. In Japan, for example, there are three writing systems: hiragana, katakana, and kanji. Of these, katakana is used for foreign words and names; therefore, name fields should not be restricted to a portion of a character set.

Honorifics and titles can appear anywhere in a name. Even in English, a name might begin with *Ms.* or *Dr.* and end with *II* or *D.D.S.*

Command Line Interface

A command line interface is often used to access a database of names. For example, commands can update an entry or search the database. These commands handle names as parameters and must accommodate the varying fields.

Character Interface

A character interface lays out name fields in a particular order. For different cultures, the number of names and the order varies. Ideally, an interface displays such information in the order most appropriate for a given locale. Field labels and their corresponding data fields change according to the requirements of the current locale.

Graphical Interface

Graphical interfaces are similar to character interfaces. They can have a more elaborate layout, but the principles are still the same.

Application Protocols

Some application protocols accommodate parts of a name. They can communicate name parts from an address book to an email field or from a directory to a search results screen.

Storage and Interchange

Not applicable.

Application Programming Interfaces (APIs)

APIs can carry out proper formatting for these items. These APIs might detect a locale or take a locale as an argument.

Requirements for Compliance

In general, name handling code must not assume that any subset of the current charset is used in a name. This is particularly important in some authentication code.

Command Line Interface

Command line parameters must be flexible enough to handle the varying name fields.

Character Interface

Character interfaces must have the flexibility to display name fields in varying layouts.

Graphical Interface

Graphical interfaces must have the flexibility to display name fields in varying layouts.

Application Protocols

Application protocols must accommodate the many name field possibilities.

Storage and Interchange

No requirement.

Application Programming Interfaces

Formatting rules for personal names, honorifics, and titles must be separate from the source code, configurable and accessible through system or proprietary locale-specific APIs.

4.2 Cultural Formatting and Processing

4.2.3.3 Addresses

Description

Addresses can vary from one culture to another. The following table provides examples of different address formats.¹

Table 4-7. Address Formats

Country	Format	Notes
China	[<Country>] <Province> <City> <Address1> <LastName> <FirstName> <Honorific>	<Province>, which is represented by two uppercase letters in parentheses, is used only if the city is not a province capital. The line with <Country> should use a negative indent.
Italy	<Title> <FirstName> <LastName> [<CompanyName>] [blank line] [<Country>]	Numbers, for example, house numbers, are always at the end of <Address> for example, via Palmanova 12. An optional blank line between <Address> and <Country> makes the address easier to read.
Russia	[<Country>] <PostalCode> [<State or Republic>] [<Region>] <City> <Address1> <Address2> [<CompanyName>] <LastName> <FirstName> <SecondName>	The <State or Republic> and <Region> fields are used only used in the following cases: <ul style="list-style-type: none"> ● Letter is sent to another state ● City is not the capital of the region, for example, Moscow Region, Zvenigorod ● Letter is sent from another state to a city that is not a regional capital, in which case both the name of the state and the name of the region are indicated, for example, Russia, Moscow Region, Zvenigorod. If <FirstName> and <SecondName> contain only initials followed by periods, it is more appropriate to include these fields on the same line with <LastName>, for example, LastName A. B.

Denmark	[<Honorific> <Title>] <FirstName> [<SecondName>] <LastName> [<CompanyName>] <Address1> <Address2> [<CountryCode>]<PostalCode> <City> [<Country>]	<p>Some countries have very special requirements if sending mail outside the country; Denmark is an example of this.</p> <p>The first and second lines can appear in reverse order, that is, <CompanyName> on the first line and <Honorific> to <LastName> on the second.</p> <p>There are two spaces between <PostalCode> and <City>. The postal code is four digits, without a separator.</p> <p>If mail is sent from abroad to Denmark, the prefix "DK-" (<CountryCode> plus one hyphen) is added to <PostalCode>.</p>
Finland	[<Title>] <FirstName> [<SecondName>] <LastName> [<CompanyName>] <Address1> <Address2> <PostalCode> <City> [<Country>]	<p>Some countries have special requirements, depending on the mail recipient; Finland is an example of this.</p> <p>The personal name appears first if the letter is of a personal nature, but the company name appears first in a business letter.</p>

1. Rhind, Graham, *Global Sourcebook of Address Data Management: A Guide to Address Formats and Data in 194 Countries*, Gower, 1999.

Command Line Interface

Addresses can be entered or returned on the command line. The command can take address values in a single order for all locales, or in a locale-specific order. Likewise, when displaying an address as output, the address can be shown according to the locale format or it might have a generic display format with labels for the different components to clarify the listing.

Character Interface

Character interfaces can use a freeform field or several labeled fields for address input. In a freeform field, the user types the address in the exact format preferred. When the address is retrieved for display, it can be returned exactly the way it was entered. However, if the individual fields are used for other purposes, for example, to enable a database search for entries from a particular city, then the fields are divided. When returned for display, they might be in the same format as the entry screen, or they might be formatted according to locale.

Graphical Interface

See "Character Interface."

Application Protocols

Application protocols handling address data can take the data as a delimited string or as individually labeled fields. They require the address to be in a specific format, which is unlikely to be locale-specific.

Storage and Interchange

Addresses are usually stored in databases, broken down into their components. In some cases, however, they might be stored as a freeform text field.

Application Programming Interfaces (APIs)

Sometimes, special APIs that use locale-specific rules are provided for address formatting. If they are available, you should use them. If they are not available, you can use product-specific APIs or pre-constructed templates for address formatting.

Requirements for Compliance

Command Line Interface

No special requirements for providers.

The following requirements apply to consumers:

- Consumers must either specify one format for all addresses, or a format for each supported locale.
- If an address is returned as output, it must either be in the locale-specific format, or in a format with labels for each of the components.
- All components necessary for the addresses of the platform-supported locales must be available, though not all are used in each locale.

Character Interface

No special requirements for providers.

The following requirements apply to consumers:

- If a user interface is used to create an address, it must be flexible enough to enable locale-specific formatting.
- No assumption can be made with regard to the amount of information required or the order of each component.

Graphical Interface

See "Character Interface."

Application Protocols

Providers must accommodate all possible components of addresses in the platform-supported locales.

Consumers must supply protocols with appropriate components and label elements, where relevant.

Storage and Interchange

Providers must accommodate all possible components of addresses in the platform-supported locales.

Consumers must supply appropriate components and locale information, where relevant.

Application Programming Interfaces

The following requirements apply to APIs:

- Ensure that formatting rules for addresses are separate from the source code, configurable, and accessible through system-sensitive or language-sensitive platform APIs.
- Provide product-specific APIs or pre-constructed templates if platform APIs are not available.
- Ensure that all characters required for addressing can be input and displayed.

4.2 Cultural Formatting and Processing

4.2.3.4 Other Formatting and Layout

Description

Computer applications process and format structured data to simplify tasks for users. For example, a business letter wizard in word processing software requests a sender and receiver address and then formats them into a business letter automatically, before the user even types a word. Applications should be aware that such formatting methods can be culture-specific. A provider is expected to have a set of interfaces to format and process structured text. A consumer deals with abstract text values. A provider also makes culture-specific formatting transparent to a consumer. For most of the formatting issues listed in previous sections, an application might just fill a consumer role. The provider might be the underlying operating system or a standard library. If an application intends to perform text formatting other than that which is available from an external provider, then it must play the role of provider as well as consumer. Applications can use the following steps to format structured data that is not listed in previous sections of this document.

1. Verify that the formatting is language-specific or country-specific. If the formatting is not sensitive to any language or country that the application is intended for, application designers can skip the following steps.
2. Identify all languages and countries that the application is intended for.
3. Define the provider interface. The interface should be abstract enough to handle all the formatting differences between languages.

NOTE: The consumer makes no assumption about format. It represents abstract data and the provider handles the formatting.

Consider, for example, the paper sizes that a printer recognizes. The application provides a choice of paper sizes in response to a request to print a document. Standard paper sizes vary from one country to another. The consumer application should be configured to print the document on any given paper size. The provider supplies all the paper sizes allowed for a given locale to the consumer. The consumer then takes the size from the provider and formats the data.

Command Line Interface

All data that a user enters is in some format. For non-technical commands, the input data is often in a locale-specific format. A locale-specific format might also be required for output data.

Character Interface

See "Command Line Interface."

Graphical Interface

In a graphical interface, the data is layed out on the screen. Many layouts are locale-specific, whether it is due to the data itself or the layout in general.

Application Protocols

Application Protocols used to transmit data often require the data to be in a specific format. The protocol can be used to parse the data, at least partially.

Storage and Interchange

For storage, a single internal format is typically used, with some form of locale indicator for data. This is to enable data formatting on retrieval. Interchange is similar to application protocols. See "Application Protocols."

Application Progammig Interfaces (APIs)

APIs take in data for processing and return the data. If the data comes in in a locale-specific format, the API must be locale-sensitive to ensure that the format is understood and the data can be correctly parsed. For output data, APIs can do either of the following:

- Publish whether the data is formatted according to the locale
- Provide one format and expect the calling application to reformat for the locale

Requirements for Compliance

The interface between the provider and consumer must be locale-independent. If an application were to be extended to handle a new formatting procedure, the provider must be able to extend the functionality without changing any interface between the provider and consumer.

Command Line Interface

Determine whether data input and output formats are locale-specific. If they are, then locale-sensitive interfaces must be built into the application. Since this type of data is less common, provider software or libraries might not be available; therefore an application must function as both a provider and consumer.

Character Interface

See "Command Line Interface."

Graphical Interface

Make layouts flexible for rearrangment based on locale. Determine whether there are sufficient fields for display for all supported locales.

Application Protocols

Allow for specification of locale, charset, or other related information necessary to correctly parse the data.

Storage and Interchange

To enable data to be retrieved and changed into the correct locale format, store locale information with the data. Interchange is similar to application protocols. See "Application Protocols."

Application Programming Interfaces

Consumers should supply locale information to providers to either accompany supplied data or to retrieve properly formatted data. Providers must provide locale-sensitive interfaces to allow for varying locale formats.

4.2 Cultural Formatting and Processing

4.2.4.1 Lexical and Grammatical

Description

The following table describes terms that are used in this section.

Table 4-8. Terms and Definitions

Term	Description
word	Basic sub-structure of a sentence which has meaning.
lexical	Relates to the words and vocabulary of a language.
morpheme	Unit of meaning which can be a word or part of a word.
morphology	Study of the structure and content of words.
grammatical	Relates to the way words are put together in a language.
syntactic	Relates to the way words are put together to form phrases or sentences.
semantic	Meaning of a sentence or word, based on the situation and the people, not just grammar.

When used in the same context, "lexical" and "grammatical" refer to language processing on a level higher than the character level. This section examines words and sentences as input and output. The following issues relate to lexical and grammatical processing:

- [Formatted sentences](#)
- [Formatted words](#)
- [Spell checkers](#)
- [Word and character count](#)
- [Word breaking and hyphenation](#)
- [Justification and orientation](#)
- [Grammar checkers](#)
- [Complex text layout \(CTL\)](#)
- [Input method framework](#)
- [Bi-directional text](#)

Formatted Sentences

Modern applications often produce message strings, such as:

```
name + "will be on the " + time + " bus to" + city
```

where *name*, *time*, and *city* are calculated somewhere else and put into a predefined sentence

structure. This is a grammatical or syntactic issue. In other languages, the order of these five items might be rearranged. The nature of the *time* variable might be different for each locale. To display sentences of this type for different locales, strings must use formatted output.

Formatted Words

Applications can process words themselves. For example, to form the plural of English words, the application can add the appropriate suffix. Other languages have different structures, for example, Arabic has singular, dual, and plural forms; China has no plurals. Words in different languages must be manipulated in different ways.

Spell Checkers

If users are writing documents in two or more languages they require spell checkers for different languages. Users might need to access multiple dictionaries at the same time, for example, Spanish and English. The dictionary should not depend on any particular encoding. Spelling checkers need to take into consideration lexical structures like suffixes or prefixes. In English, for example, a spelling checker might remove the prefix *un-* from a word before looking it up. For information on issues that affect other languages, see "[Word Breaking.](#)" The spell checker must be generic enough to handle these issues.

Word and Character Count

Some languages do not have space boundaries and characters can be compound or multi-byte. Chinese, for example, presents word count problems, as characters are usually counted. For counting words, a dictionary look-up mechanism is required.

Word Breaking and Hyphenation

In English, word boundaries are marked by white space. This is not true in other languages, such as Thai. While breaking along word boundaries can be accomplished using dictionary look-up or white space delimiters, breaking in the middle of a word for hyphenation is more difficult. For some languages, it might not be a serious problem if a word is broken in the wrong place. In complex text layout (CTL) languages, however, it can change the way the word is rendered. Consider, for example, bi-directional languages and multi-language texts. Users might not always work in a single language. In Chinese, hyphenation is not a problem, because words can be broken along any character boundary.

Justification and Orientation

Some languages, like Arabic and Hebrew, are bi-directional languages. Numbers are read from left to right, words are read from right to left. If there is left-to-right language text, it should be read from left to right. Think about how things are justified. Chinese and Japanese are often written vertically and parentheses and punctuation are re-oriented. Embedded text in other languages is also rotated in vertical writing.

Grammar Checkers

Grammar varies widely from language to language. In some languages, every word in a sentence has a contextual suffix, in others, suffixes are never appended. When designing and coding a grammar checker, use the greatest common denominator. If you have word class files, make them generic so that grammar checkers for other languages can be used. Have a parser that acts as a base engine with a plug-in module for each supported language.

Complex Text Layout (CTL)

CTL is a lexical issue. Text can be input, which is based on sound, character shapes, or even character position. From the input, a character output can be generated in the form of single characters, groups of characters, or words. As more input is received, the output can change based on the new context. For more information, see section [4.3.3.3](#).

Input Method Framework

In the past, input method frameworks were considered a character issue. For Chinese, Japanese, and Korean, input methods took several characters as input and produced a single character output. Now they are, in reality, a lexical issue; they can take character input and produce multi-character or word output. They are responsible for managing [CTL](#). For more information, see section [4.3.3.2](#).

Bi-directional Text

Bi-directional processing, sometimes known as *bi-di*, is also a lexical issue. The system must know what kind of morpheme it is dealing with, so that it can justify and orient it in the right direction. In the case of a Hebrew string, which includes a price and product name in Latin script, the text begins on the right side of the screen for the Hebrew text and continues right to left. Once the Latin text is encountered, the direction switches from left to right. For rendering, this means displaying the first character of the Latin text, then moving it to the left. The next character is then displayed to the right of it. This continues until more Hebrew text is encountered. So, using uppercase for Hebrew and lowercase and numbers for Latin, the text "THIS IS HEBREW this is latin \$10.95 MORE HEBREW." goes through the following stages:

```

...
WERBEH SI SIHT
t WERBEH SI SIHT
th WERBEH SI SIHT
...
.this is latin $10 WERBEH SI SIHT
this is latin $10.9 WERBEH SI SIHT
...
.WERBEH EROM this is latin $10.95 WERBEH SI SIHT

```

For the data stream, however, characters are in logical order:

```

Data comes in from this direction ==>
.WERBEH EROM 59.01$ nital si siht WERBEH SI SIHT

```

Command Line Interface

The command line accepts lexical and syntactic data as input and returns lexical and syntactic data.

Character Interface

A character interface, like the command line, takes character input and produces character output. For lexical issues, the boundary between character and word is blurred. This can involve input method editors for languages, such as Chinese, Japanese, and Korean, and

complex text layout (CTL) for languages, such as Tamil and Thai. CTL is part of the morphological level as well as the character level.

Graphical Interface

For graphical interfaces, lexical and grammatical data can be layered onto graphical objects, adding a layer of complexity to lexical and grammatical handling.

Application Protocols

Protocols can include strings of words as part of the protocol stream or identify sentence data.

Storage and Interchange

Storage and interchange formats can encapsulate word or sentence data.

Application Programming Interfaces (APIs)

APIs can take lexical or syntactic data as parameters to calls and return lexical or syntactic data.

Requirements for Compliance

In general, providers must supply functions that can accommodate the lexical and syntactic manipulation needs of the consumer. Consumers must use provider functions and manage the word and sentence structure so as to accommodate data in any of the provider locales. This means that the consumer must supply the provider functions with required information for correctly processing the data.

Command Line Interface

Providers must supply parsing functions for reading in and returning lexical and syntactic data to the command line in any language they support.

Consumers must use provider supplied lexical and syntactic functions, making sure to accommodate the lexical and syntactic structures of other languages.

Character Interface

Providers must supply input method (word forming) functions for reading in and returning word and sentence data to the character interface in any character set they support.

Consumers must use provider supplied input method (word forming) functions, making sure to accommodate multi-byte characters for input and output, as well as single byte.

Graphical Interface

Providers must supply lexical functions for managing character data with various elements

of the graphical user interface (GUI), such as buttons, drop-down lists, and title bars. These functions must accommodate all supported character sets.

Consumers must use provider functions for creating the GUI, supplying language data for proper handling.

Application Protocols

Providers must construct the protocol so as to accommodate the necessary lexical data in a specified format.

Consumers must implement the protocol with all related character information, including charset, language, and locale.

Storage and Interchange

Providers must allow for storage of any lexical and syntactic data, supplying formats that contain relevant information for proper retrieval.

Consumers must include all relevant information in the storage and interchange formats so that lexical and syntactic data in any charset can be properly retrieved.

Application Programming Interfaces

Providers must supply interfaces that accommodate any lexical and syntactic data, where relevant.

Consumers must supply relevant lexical or semantic data descriptions to the API functions to properly process lexical or syntactic data.

4.3 Text Foundations and Writing Systems

4.3.3.3 Device Output

Description

Before describing device output, it is important to draw a distinction between characters and glyphs. A character is an abstract representation used for text processing. A glyph is a physical shape that is used to represent a character. There is not necessarily a 1-to-1 mapping between characters and glyphs. In general, there is an m to n mapping between characters and glyphs where $m > 0$ and $n > 0$. For example:

ü = u + umlaut (diacritical mark), 1 character to 2 glyphs

f + f = ff (ff ligature), 2 chars to 1 glyph

For more information on characters, see [Section 4.3.2.1 "Characters \(Semantics and Codespaces\)."](#)

This situation is even more common for languages that require what is collectively referred to as complex text layout (CTL), for example, Thai. In Thai, consonants can combine with vowels and tone marks to be represented on the screen as a single glyph.

The glyph point size needed to ensure legibility of characters tends to vary between languages. While a Latin based language can render quite well with an 8 point font, Chinese might need a much larger font size before the glyphs are readable on the screen.

Another aspect of character output is the direction of character rendering on the output devices. The writing system is not always left-to-right and top-to-bottom. In Arabic and Hebrew, the text can be bi-directional, going from right-to-left mixed with left-to-right. In Japanese text processing systems, rendering text vertically from top to bottom is very common.

It is important that software products make no inherent assumptions about the mapping between characters and glyphs and the directionality of the text.

Software that controls the display by carrying its own graphics kernel, rasterizer, and fonts are generally considered a provider. If an application provides command line interface or character based interface, it does not control the display properties, such as fonts or color, directly. Instead it controls the appearance indirectly by using terminal interface libraries, such as `libcurses`). To do this, these terminal libraries interact with terminal emulators, such as `dtterm`). Hence software that provides a terminal interface is also considered to be a provider, as it is responsible for displaying application messages and obtaining user input. The provider software exports its display services either natively or using higher level interfaces, such as Motif in UNIX or Java[tm] Foundation Classes/Swing.

Command Line Interface

Providers receive text in the form of character encoding values and render characters according to the proper encoding and locale. The proper encoding and locale can be determined using system queries or by parameters included along with the text. To supply

the text, and possibly encoding and locale information to the provider, consumers call provider functions .

Character Interface

In addition to proper character rendering, providers allow for text wrapping and spacing variations. Consumers call the provider functions with charset and locale information.

Graphical Interface

Graphical interfaces involve not only correct character rendering and text wrapping, but also formatting of layout. Text is more likely to be produced in a proportional font, juxtaposed with graphics. Objects such as buttons, scroll bars, and input fields must be managed according to the size of the rendered text, the orientation, and the input method.

Application Protocols

Not applicable.

Storage and Interchange

Not applicable.

Application Programming Interfaces (APIs)

Provider APIs can supply character-to-glyph transformation, font metrics data, string rendering and text wrap, and layout orientation swapping. Consumers provide the data stream, charset, locale, size, and other required information using the APIs.

Requirements for Compliance

Command Line Interface

Providers must map character codes into the correct corresponding glyphs.

Consumers must supply provider functions with the required information for correct rendering.

Character Interface

In character interfaces, provider software should handle character-to-glyph mapping, shielding consumer applications from dealing directly with glyphs. In addition, providers should handle plain text wrapping and spacing based on the character set and language.

Consumers must supply provider functions with all necessary information to correctly render the data.

Graphical Interface

The following requirements apply to providers:

- Providers should not make any assumptions about layout or directionality of components.
- Providers must be able to accommodate complex text layout (CTL) rules, in both horizontal and vertical directions.
- Providers should not optimize rendering based on a single glyph.
- Font metrics computation should be flexible enough to take into account multiple characters per glyph, zero width characters, and composite characters.
- Providers should not assume that a single physical font contains all the glyphs. Glyphs can be found in a set of fonts logically grouped together.
- Characters and glyphs must be handled in a locale independent way.

The following requirements apply to consumers:

- Consumers must supply provider functions with the information necessary for proper output.
- Consumers must not break compliance by overriding provider facilities or embedding character-to-glyph mapping and directionality.

Application Protocols

No requirement.

Storage and Interchange

No requirement.

Application Programming Interfaces

APIs must contain enough information to enable providers to correctly transform characters into glyphs, retrieve accurate font metrics, render strings, and wrap text.

If possible, consumers should use APIs provided by the underlying platform to convert characters into glyphs and to calculate font metrics.

4.3 Text Foundation and Writing Systems

4.3.2.1 Characters (Semantics and Codespaces)

Description

The following table describes terms that are used in this section. Several of the definitions are taken from [RFC 2130](#) - *The Report of the IAB Character Set Workshop*, 29 February - 1 March, 1996.

Table 4-9. Terms and Definitions

Term	Definition
character	General representation of a single written symbol used in a writing system. This can include symbols, punctuation, and in computer terms, control codes.
character set	Complete group of characters for one or more writing systems. More complete than an alphabet.
glyph	Graphical representation of a character. For example, the character "LATIN SMALL LETTER A" can appear as the glyphs "a", "a", "a", and "a."
coded character set	Mapping from a set of abstract characters to a set of integers.
codeset	See coded character set.
character-set-name	Official or unofficial name used to refer to a codeset.
charset	Name used to refer to a defined computer character set standard.
character encoding scheme	Mapping from a coded character set (or several) to a set of octets.
transfer encoding syntax	Transformation applied to data that has been encoded using a character encoding scheme to allow it to be transmitted.
single-byte	Data with a value of length 1 byte, or 8 bits.
multi-byte	Data with a value of varying length from 1 byte, or 8 bits, to 6 bytes, or 48 bits.

A character has no fixed semantics; that is, characters change their behavior depending on the context. Consider the following aspects of character semantics as they relate to program code:

- [Production](#)
- [Size](#)
- [Classification](#)
- [Equivalence](#)

Production

Most English letters are produced using a single keystroke, but to produce the ligature **æ**, several keystrokes are required. For Asian languages, even more keystrokes might be necessary to produce the desired glyph. Several characters might be necessary to form a single glyph, as in Korean Hangul.

Size

A glyph for a particular character can vary in size and shape from typeface to typeface and language to language. For example, here is the character **w** in several typefaces: **w**, *w*, **w**, *w*. To illustrate a language context, Polish accent marks are closer to their base letters than French accent marks.

Classification

Some languages may consider glyphs as uppercase and lowercase of a single character, some others categorize them by their position in a word. Languages written in Latin, Cyrillic, and Greek scripts have case distinction, those written in Arabic have standalone, initial, medial, and final forms.

Equivalence

With the differences in classification come differing rules for equivalence. Even among different users of the same language there are different concepts of character equivalency.

Command Line Interface

Command line reads in characters in the form of commands and their parameters and returns characters in the form of data. While commands themselves are to remain constant regardless of which localized product they are in, the parameters might be data in any codeset.

Character Interface

A character interface, like command line, takes character input and produces character output.

Graphical Interface

For graphical interfaces, characters can be layered onto graphical objects, adding a layer of complexity to character handling.

Application Protocols

Protocols can include character data as part of the protocol stream or identify character data.

Storage and Interchange

Storage and interchange formats usually accommodate character data.

Application Programming Interfaces (APIs)

APIs can take character data as parameters to calls and return character data.

Requirements for Compliance

In general, providers must supply functions that can accommodate any character encoding scheme. Consumers must use provider functions and manage the codesets so as to accommodate data in any of the provider codesets. This means that the consumer must supply the provider functions with required information for correctly processing the data.

Command Line Interface

Providers must supply character functions for reading in and returning character data to the command line in any character encoding scheme they support.

Consumers must use provider supplied character functions, making sure to accommodate multi-byte characters for input and output, as well as single byte.

Character Interface

Providers must supply character functions for reading in and returning character data to the character interface in any character encoding scheme they support.

Consumers must use provider supplied character functions, making sure to accommodate multi-byte characters for input and output, as well as single byte.

Graphical Interface

Providers must supply character functions for managing character data with various elements of the graphical user interface (GUI), such as buttons, drop-down lists, and title bars. These functions must accommodate all supported character encoding schemes.

Consumers must use provider functions for creating the GUI.

Application Protocols

Providers must construct the protocol so as to accommodate any character data in some specified format.

Consumers must implement the protocol with all related character information, including charset, language, and locale.

Storage and Interchange

Providers must allow for storage of any character data, supplying formats that contain

relevant information for proper retrieval.

Consumers must include all relevant information in the storage and interchange formats so that character data in any character encoding scheme can be properly retrieved.

Application Programming Interfaces

Providers must supply interfaces that accommodate any character data, where relevant.

Consumers must include relevant character data descriptions to the API functions to properly process character data.

Copyright 1994-2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

4.3 Text Foundation and Writing Systems

4.3.3.2 Device Input (Keyboard and Input Methods)

Description

You can use a variety of devices to input data into a computer system. These devices include:

- Keyboard
- Mouse
- Light pen
- Optical character recognition (OCR)
- Software combined with a scanner
- Voice recognition

Different platforms support different input devices. They also support different methods for the user to input information. These different styles of user input are generically referred to as input methods (IMs). Input methods are usually mentioned in association with Asian languages, but less complex ones are also used to input English and European language characters. IMs are often provided by the computer system vendors, but some are developed and distributed by third party vendors.

The software used to implement an IM can consist of several different sub-components. This section describes the following sub-components:

- [Input device driver](#)
- [Input device driver interface](#)
- [Front-end processor \(FEP\)](#)

Input Device Driver

This is the software used to transmit and receive the raw input from the peripheral device. This software also processes the raw input and maps it into a data representation that can be processed by the computer system and application software.

Input Device Driver Interface

This interface implements the interaction between the input device driver and the operating system. The definition of a standard set of interfaces enables the development of device drivers from a variety of suppliers.

Front-End Processor (FEP)

An FEP is usually associated with the input of languages with large character sets, such as Japanese or Chinese. An FEP implements intermediate processing of characters to map abstract input data--for example, multiple key-strokes--to a unique character representation and deliver that character to the software product.

Command Line Interface

Software products can provide command line interfaces that allow the user to specify the input style of the input methods (IMs). These command line interfaces are dependent on the software products, the platforms of the computer systems, and the IMs that are being used. Command line can also work with IMs, enabling the user to input complex character data.

Character Interface

It is rare to see a character-based interface working with complex IMs. They function similarly to command line input.

Graphical Interface

Front-end processors (FEPs), which are used to input languages with large character sets, usually provide graphical user interface (GUI) windows for end-user interaction. These types of windows include:

- **Status window** - This window displays the current interaction status of the FEP. For example, a window that you use to enter Japanese kana characters can have a kana status displayed. To change the interaction status, you can type a sequence of keystrokes that communicate the status change to the FEP.
- **Conversion/Composition window** - This window displays the characters that you type before any FEP conversion, for example, the hiragana characters that you use to phonetically write a Japanese word before the FEP converts the input into an ideographic character.
- **Candidate window** - When an FEP converts input data to the final display form, it might not be able to uniquely identify the display character. When this happens, the FEP displays a list of candidate characters. Candidate characters are the set of display characters that the FEP can determine as possibilities based on user input. You can select the display character that you require from the candidate list.

FEPs provide several interaction options for the end-user. These options include:

- **Root window style** - This input style moves all of the responsibility for large character set handling to the FEP. The FEP intercepts the input data and opens a window independent of the software product, usually at the bottom of the display screen. This separate window is used to display the FEP interaction status, conversion/composition characters, and candidate characters. When the desired display character is selected, the FEP returns that character to the software product.
- **Off-the-spot** - This input style is similar to root window, except that the FEP window can be located at any position on the software product window. This is more efficient for the end-user than the root window style.
- **Over-the-spot** - This input style is similar to root window, but the FEP window is located at the same position that the display character is entered on the software

product window. This is more efficient for the end-user than the off-the-spot style.

- **On-the-spot** - This input style requires much more active participation from the software product. The software product is responsible for displaying conversion/composition characters at the point where the final display characters are located. It must coordinate with the FEP to determine when conversion will occur. It must then accept the display character from the FEP and display it in the software product window.

Application Protocols

Not applicable.

Storage and Interchange

Not applicable.

Application Programming Interfaces (APIs)

Software products use two types of APIs for processing inputs from device. These are the device driver APIs and the APIs to communicate with FEPs.

The APIs for device drivers provide the software product with information about the abstract data being input. They also provide methods to obtain detailed information about the input data.

The APIs for input method (IM) engines enable the software product to control the input style of the IM. These APIs also provide the ability to control how the IM processes and delivers the input data to the software product.

Requirements for Compliance

For the processing of device input to be fully internationalized, the software product must be able to process data in any supported language, no matter which encoding scheme is used. The software product must also be able to interact with front-end processors to accept characters from certain languages.

Command Line Interface

Providers must allow terminal window input and supply an interface that can be used by command line consumers.

To enable users to input any kind of character data on the command line, consumers must be able to handle output from an FEP.

Character Interface

Similar to command line, providers must be able to supply final character data to a character interface window.

Consumers must call the necessary provider functions to enable input using an input method (IM) editor.

Graphical Interface

Providers must supply full IM functionality for at least root window input. They should provide the following input style options to the consumer:

- [Root window](#)
- [Off-the-spot](#)
- [Over-the-spot](#)
- [On-the-spot](#)

Consumers should enable the user to select any of these input style options. Consumers must accept the output of the IM editor.

Application Protocols

No requirement.

Storage and Interchange

No requirement.

Application Programming Interfaces

Device driver APIs must be supplied for devices commonly used in the locales where the product is sold. IM APIs must be sufficiently flexible to handle all the major characters and character sets of the world.

4.2 Cultural Formatting and Processing

4.2.4.2 Phonology and Sound-to-Text

Description

This section examines emerging and developing technologies in language processing, including:

- [Machine Translation](#)
- [Speech-to-text](#)
- [Text-to-speech](#)
- [Pen-to-text](#)

Machine Translation

Machine translation is still a developing technology and most functional systems are based on two language pairs. A system that takes internationalization into account might have a generalized parser that can be easily modified for other language pairs. One approach is to translate into an intermediate structure sometimes called a *deep structure*, and then a translate-in program and a translate-out program can be written for each natural language. The system would be internationalized and not dependent on a language pair.

Speech-to-Text

Most speech recognition systems today operate on a lexical system. They recognize an utterance, but cannot parse connected speech. Some of the more modern ones can recognize connected speech and thus deal with grammatical structure. For a connected speech product, a general parsing engine with plug-ins for supported languages constitutes a good, internationalized design.

Text-to-Speech

Some text involves multiple scripts and languages, and so needs access to multiple dictionaries at the same time. For phonetic output, super-segmental conditioning such as intonation might need to be included.

Pen-to-Text

As in text-to-speech, some text contains several scripts and languages and requires multiple dictionaries at once. For some scripts, stroke direction and order are significant and can aid in determining the character written. Other scripts focus on completed character shape; therefore, a method for determining character completion must be established.

Command Line Interface

Command line interfaces can read pen or speech based data in and return text data.

Character Interface

A character interface, like command line interfaces, take speech or pen input and produce

character output.

Graphical Interface

For graphical interfaces, speech or pen data can be layered onto graphical objects, adding a layer of complexity to the usual graphical interface handling.

Application Protocols

Protocols can include pieces of speech as part of the protocol stream.

Storage and Interchange

Storage and interchange formats can encapsulate speech or pen data.

Application Programming Interfaces (APIs)

APIs can take speech data as parameters to calls and return speech data.

Requirements for Compliance

In general, providers must supply functions that can accommodate the lexical and syntactic manipulation needs of the consumer. Consumers must use provider functions and manage the word and sentence structure, so as to accommodate data in any of the provider locales. This means that the consumer must supply the provider functions with the required information for correctly processing the data.

Command Line Interface

Providers must supply speech recognition functions for inputting and returning lexical data to the command line in any language they support.

Consumers must use provider supplied speech or pen recognition functions, making sure to accommodate the speech or script structures of other languages.

Character Interface

Providers must supply speech or pen input method functions for inputting and returning data to the character interface in any character set they support.

Consumers must use provider supplied speech or pen method functions, making sure to accommodate multi-byte characters for input and output, as well as single byte.

Graphical Interface

Providers must supply speech or pen recognition functions for managing character data with various elements of the graphical user interface (GUI), such as buttons, drop-down lists, and title bars. These functions must accommodate all supported character sets.

Consumers must use provider functions for creating the GUI, supplying the required

language data.

Application Protocols

Providers must construct the protocol so as to accommodate the necessary speech or pen data in some specified format.

Consumers must implement the protocol with all related character information, including charset, language, and locale.

Storage and Interchange

Providers must allow for storage of any speech or pen data, supplying formats that contain relevant information for proper retrieval.

Consumers must include all relevant information in the storage and interchange formats so speech or pen data in any language can be properly retrieved.

Application Programming Interfaces

Providers must supply interfaces that accommodate any speech or pen data, where relevant.

Consumers must supply relevant speech or pen data descriptions to the API functions to properly process speech or pen data.

4.3 Text Foundation and Writing Systems

4.3.1 Writing System Negotiations, Defaults, and Selection

Description

Software processes data. If the text is data, the most basic task for the software is to identify the writing system to which the text belongs. In other words, a program must know the character set of the textual data in order to properly interpret the bytecodes. In the past, programs assumed the data to be in a particular character encoding scheme. This assumption is not possible for a global software product. Programs must find out what character encoding scheme is being used, and from there, the writing system can be determined. Given no information, it is not possible to determine the character set by programmatically inspecting the bytestream.

For the most part, knowing the character encoding scheme is enough for rendering purposes. In some cases, further determination must be made from the code points used to correctly render the text. For example, if the UTF-8 character encoding scheme is used to represent Arabic text, the rendering process must first know that the text is in UTF-8, and then check the code points actually used in the text. Once it finds that the code points are in the Arabic range, a special Arabic rendering process must be called, since Arabic has a range of shapes for each code point depending on word position.

The method for figuring out which character encoding scheme or charset is being used is different for particular situations. For example, in HTML forms, the charset of the input data is the same as that set for the page; however, if the charset of the page is not explicitly declared in an HTTP header or META tag, the data might be in the default charset set for the browser. For files on a Solaris machine, the data can be in any charset supported by Solaris. In databases, the charset is usually defined. The best situation is for the client to force the data coming from the user into a particular charset and then convert it to Unicode. The server can then use Unicode internally; however, this is not always possible.

Both Solaris and Windows locales have associated charsets and can be queried for the currently active charset. Many interfaces depend on the locale charset and use a system call to find out which one is active.

Command Line Interface

Commands take in textual data as parameters and return textual data. The charset of this data can be the system locale charset or one specified in another command parameter. Rendering is handled by the terminal window, which determines the necessary font glyph associations. In some situations, text is not expected to be rendered correctly, for example, text which is inserted into a database, and which is normally retrieved through a graphical interface. A user might use a terminal that does not render that particular charset, but the bytecodes can still be entered.

Character Interface

The rendering system needs to handle the charset used in the character interface, and so must be informed. In addition, input fields in the interface are in a charset that must be determined.

Graphical Interface

Graphical interfaces usually have text that can be displayed in some charset, as well as text input areas. Several mechanisms in graphical interfaces allow for explicit charset identification. Rendering is much more sophisticated due to the more flexible nature of graphics. A variety of font options are available. These can be set by the application or by the user.

Application Protocols

If an application protocol includes textual data, then either the protocol specifies what charset the data is in, such as in LDAPv3, or the protocol contains a charset parameter, such as in HTTP.

Storage and Interchange

Storage and interchange are similar to application protocols; either they are defined as having text in a particular charset, or they allow for charset specification.

Application Programming Interfaces (APIs)

APIs can use the system charset, require a particular charset, or allow a parameter to specify the charset. Which one is used depends on the functionality provided.

Requirements for Compliance

In all cases, the charset of the text must be unambiguous.

Command Line Interface

Providers must state the method of charset determination. They should provide as much flexibility as functionally needed, not always relying on the system charset.

Consumers must be aware of the provider charset determination method and supply a charset parameter, where relevant.

Character Interface

See "Command Line Interface."

Graphical Interface

Providers must allow for charset specification by the consumer.

Consumers must supply providers with the text charset.

Application Protocols

Providers must either allow for a charset value or clearly state the assumed charset. Recommend using an encoding of Unicode if only a single charset is allowed.

Consumers must supply the charset value or convert the data into the assumed charset before using the protocol.

Storage and Interchange

See "Application Protocols."

Application Programming Interfaces

Providers must state the method of charset determination. They should provide as much flexibility as functionally needed, not always relying on the system charset.

Consumers must be aware of the provider charset determination method and supply a charset parameter, where relevant.

4.3 Text Foundation and Writing Systems

4.3.2.2 Strings (Encoding Methods and Transcoding)

Description

Strings are the primary text element processed by software. Programs need to:

- [Determine string boundaries](#)
- [Calculate string length](#)
- [Compare strings](#)
- [Move strings from one place to another](#)

To Determine String Boundaries

Finding the beginning of a string is fairly straightforward; essentially the first byte in a given parameter, variable, or object can be safely assumed to be the start. However, determining where the string ends is much more difficult. You can do this in any of the following ways:

- **Look for a null (x'00') byte** - This is effective in most cases, but there are certain Unicode character encoding schemes (UTF-16, UCS-4, UCS-2) that contain nulls as part of a character.
- **Use a length value** - A pre-determined length value can be provided as an additional parameter, dictating the number of bytes in the string.
- **Look for a particular delimiter** - While similar to inspecting for a null byte, this tends to be more difficult to implement. If the delimiter is a single byte value outside of the range x'00'-x'7F', it can appear as part of a multibyte character in many character encoding schemes. Even a delimiter in the range x'20'-x'7F' is embedded inside characters in several 7-bit encoding schemes and the entire 7-bit range x'00'-x'7F' is used to make up multibyte Unicode characters in UTF-16, UCS-4, and UCS-2.
- **Find a language-related punctuation mark or whitespace** - For certain text processing products, this method of boundary determination is basic functionality; However, this requires a tremendous amount of supporting information, including language, charset, punctuation mark byte sequences per language/charset combination, and more. Programs must also handle the parsing of textual data in different charsets.

If textual data is restricted to a certain charset, then it is possible to look for a particular delimiter.

To Calculate String Length

String length can mean two different things: physical length in bytes and conceptual length in characters. Both concepts of length are important to string handling. It is for the specific

application to determine which length is needed at any given point in the program. For applications that do not actually process the individual characters of a text string, length in characters is probably not useful.

To Compare Strings

Strings are compared for a number of purposes. An input string can be matched against a list of actions to determine whether a task has been initiated. User-entered search strings are compared against a body of text to find matching data. Strings are collated based on the results of a comparison.

In order to successfully compare two strings, they must be in the same charset. Some programs work with a restricted set of charsets, such as those covering the Japanese scripts. Both strings should be converted to the common charset, if they are not already in it. In this context, the common charset should be the one that is a superset of all the possible charsets. For software set up to work with any of the major charsets in the world, it is safest to choose an encoding of Unicode, such as UTF-8 or UTF-16, as the common charset.

Unicode needs special processing, however, due to its ability to represent the same character in several different ways. For example, the character ü can be U+00FC or the combination U+0075 U+0308, but to a user, the character is the same and should always match, regardless of the underlying values. To achieve the expected results, Unicode data must be *normalized*; that is, only one of the representations for each character is allowed and the data is converted to that set of representations. Obviously, there is more than one way to normalize the text, for example, the representation chosen for ü could be either the single value U+00FC or the combination of values U+0075 U+0308. Unicode contains definitions of different normalization forms. A program uses only one of the forms throughout its processing. For more information on Unicode normalization, see the [Unicode Technical Report #15](#).

Sometimes data requires another type of processing called *canonicalization*.

Canonicalization is needed in situations where two different characters must compare the same. An example is changing all the characters to lower-case for case-insensitive matching. Not all writing systems have case, but there are many different forms of canonicalization. In Hebrew for example, certain accents and points can be ignored for comparison.

To Move Strings From One Place to Another

Programs often move strings from one place to another. For example, user input strings are retrieved and stored in a database. In some cases, the text must be converted from one character encoding to another during this process. If a product is handling data in all the major charsets, it makes sense to store and process data in a Unicode encoding. So when the data is retrieved from the user interface, it is first converted from its original charset to a Unicode encoding, and then stored in a database.

Usually it is not enough to simply convert a string into a Unicode encoding and store it. If a user wants to retrieve the string for viewing and does not have a configuration that supports the display of Unicode encodings, the string must be converted into an appropriate charset. It might not be necessary to convert it back into its original charset, but it is important to know what charset can support the characters in the string. Either the original charset, or more commonly, the language of the data, should be stored with the string.

Command Line Interface

Strings can be parameters to the command on the command line. They can be in files that are taken as input or typed in directly. They can also be returned as part of executing the command. Delimiters in this case are usually whitespace, though if the strings are contained within a file, they could be delimited with some other designated character. String data can be restricted to an encoding of Unicode or the default charset of the locale for the terminal window. For more information, see [Section 4.3.1](#).

Character Interface

String data can be input into a character interface. The data is probably in the default charset of the current locale, and needs to be handled accordingly (see [Section 4.3.1](#)). If the string is limited to a specific byte length, special processing might be necessary to ensure that only entire character values of multibyte characters are read into the string buffer. Output strings need more room for display in other languages and any display length truncation is done at character divisions. Sorted output is displayed in the logical order for the locale.

Graphical Interface

Similar to the character interface, graphical interfaces take strings as input data. Usually, graphical interfaces have more control over the charset of the input data. The length, delimiter and sort issues are the same as in character interfaces.

Application Protocols

Protocols are used to transport strings from application to application. They either allow for specification of the charset of the string data within the protocol, or require the data to be in a specific charset. Delimiters are also defined in the protocol and should accommodate all allowed string data.

Storage and Interchange

Strings can be stored in their entirety or parsed into relevant pieces and stored. Either they are converted and stored in a single specified charset (Unicode encoding) with language information, or they are stored along with their charset identifier. The storage format specifies a delimiter. File formats accommodate string data in a similar way, either forcing a Unicode encoding or taking the data in the locale encoding. Most file formats do not include charset information; this is managed external to the file.

Application Programming Interfaces (APIs)

APIs specify string delimiters or a length. They can handle strings in all different charsets or restrict them to a specific one. If the API converts from a charset into a Unicode encoding, it should follow one of the normalization forms in the [Unicode Technical Report #15](#). Canonicalization can also be performed by the API and is part of the specification.

Searching is conducted by the API on the normalized and in some cases canonicalized strings.

Requirements for Compliance

For all interfaces, providers must not truncate strings in the middle of a multibyte character. They must normalize string data that is to be compared.

For all display interfaces, consumers must display, search, and sort results in the locale-specific order of the user or if that is not available, in the order of the language or locale of the string data itself.

Command Line Interface

Providers must specify whether a Unicode encoding or locale charsets are accepted. If locale charsets are accepted, then all supported locale charsets must be accepted. Providers must specify the string delimiters used.

Consumers must supply string data in the charsets accepted by the provider. They must use the proper delimiters and adhere to length limits set by the provider. If needed, they must include charset information.

Character Interface

Providers must accept string data in the locale charset.

Consumers must have sufficient space for full string display or a meaningful string truncation.

Graphical Interface

Providers must specify whether a Unicode encoding or locale charsets are accepted. If locale charsets are accepted, then all supported locale charsets must be accepted.

Consumers must ensure that displayed strings are not truncated, or if necessary, are truncated in a meaningful position.

Application Protocols

Providers must accommodate string data in supported locale charsets with charset identification, or specify a Unicode encoding, and where relevant, the language of the string data.

Consumers must supply provider protocols with strings encoded in the appropriate charset and charset descriptions or language data, where allowed in the protocol.

Storage and Interchange

Providers must be able to store strings either in all supported charsets, or in a specified Unicode encoding. They must provide a mechanism for associating charset or language information in the case of storage.

Consumers must parse strings on appropriate boundaries, characters, words, or phrases, whichever is relevant. They must include charset or language information as necessary for proper processing and retrieval. They must externally manage charset information when it is not internal to the interchange format.

Application Programming Interfaces

Providers must specify whether strings are to be in platform supported charsets or in a Unicode encoding. They must have parameters for charset or language information, where relevant. If converting from a charset into Unicode, they must follow one of the normalization forms in the [Unicode Technical Report #15](#) and specify which one. If search functions are provided, they must specify whether fuzzy matching is performed and should allow for some configuration of fuzzy matching. If canonicalization is performed, the exact method must be specified.

Consumers must provide charset or language information as necessary. Where relevant, they should specify the type of fuzzy matching in a search.

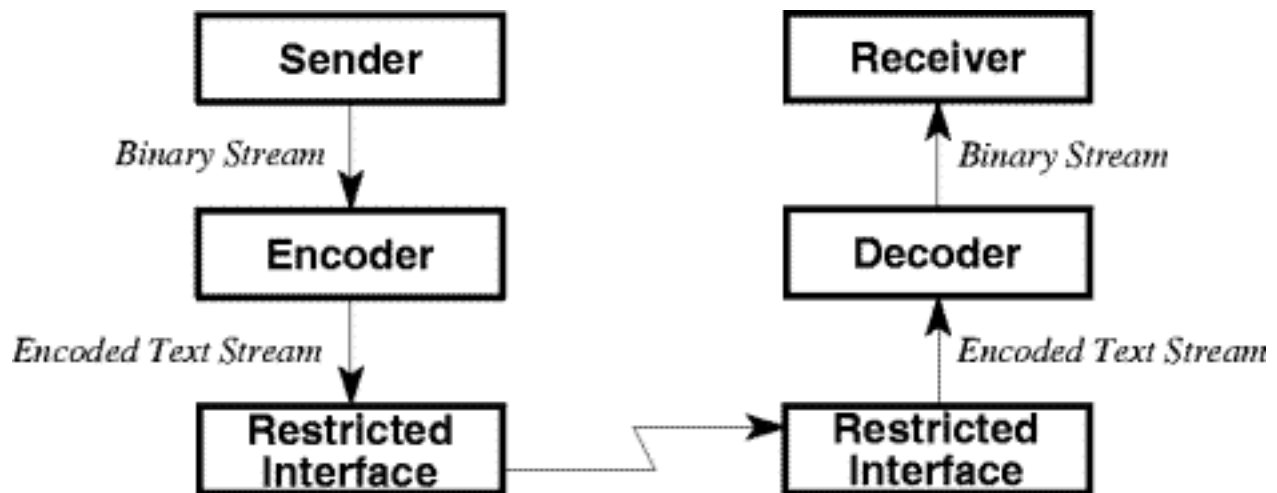
4.3 Text Foundation and Writing Systems

4.3.3.1 Transfer Encoding (8-Bit Clean)

Description

A transfer encoding is a reversible transformation that maps a data set containing a wide range of bytes to and from a restricted set of bytes. For example, a transfer encoding can map a data set of 8-bit text to 7-bit text and vice versa. Transfer encoding is used to create a "tunnel" between two cooperating applications, which enables them to exchange data bytes that would otherwise be discarded or corrupted by the interface between them. The transfer encoding is applied to the data stream before it is sent to the interface. The transfer encoding is then removed or decoded when retrieved from the interface. The following diagram shows an overview of transfer encoding.

Figure 4-1. Transfer Encoding



The most common use of transfer encodings is to send binary or 8-bit data over a communications channel that only supports 7-bit text. For example, the 8-bit data might include JPEG images, audio streams, and international text. Transfer encodings are also used to protect the data against unwanted modifications made by the interface, such as line wrapping or whitespace normalization.

Transfer encoding is a distinct layer with its own interfaces, properties, and configuration. The use of a transfer encoding layer is to be avoided whenever possible, because it introduces complexity, often has to be managed out of band, and reduces bandwidth because it increases the size of the data set. Many communication protocols consider the need to operate over *restricted* interfaces a part of their design, and so avoid the need for a transfer encoding layer. Many codesets have also been defined to be compatible with specific well-known interfaces, and likewise avoid the need for transfer encoding. In general, the only time a transfer encoding is necessary is when a legacy interface is being called upon to deliver a new application or data type that the interface was never intended to handle. This section examines the following concepts:

- [Signaling](#)
- [Idempotency](#)

- [Transfer encoding data flow](#)

Signaling

Historical transfer encoding mechanisms have not provided any reliable means for the sender to signal the receiver that a transfer encoding was being used. Instead, ad hoc signaling mechanisms were used. For example, if a user received an email message that contained the word `begin` followed by a filename, followed by many lines of gibberish, it could reasonably be inferred that the message contained an attachment that had been encoded using `uuencode`. When setting up a Usenet connection, the two administrators would agree on which compression techniques and transfer encodings they would use for their specific connection. If the sender and receiver did not agree on the transfer encoding to be used, the receiver software would either deliver garbage or return an error diagnostic.

The Multipurpose Internet Mail Extensions (MIME, RFC 2045) is one of the few cases that provides a robust signaling mechanism for the use of transfer encoding. Each body part within a MIME message contains a header field, which specifies the transfer encoding that has been applied to that part. Implementors are encouraged to pick the encoding that yields optimal results for their original data. Optimal is usually defined as the encoding type that results in the least growth in the size of the body part.

Idempotency

All known Internet transfer encodings are not idempotent; that is, if you apply the transfer encoding to a stream that has already been encoded, the stream is encoded again and must be decoded twice to obtain the original data.

Transfer Encoding Data Flow

The following steps describe a complete transfer encoding data flow:

1. The sender opens the interface to the encoder. If the encoder supports more than one type of encoding, the sender can specify which type of encoding to use or allow the encoder to choose an optimal type. It is important that the sender is able to override the encoder in selecting an encoding type, since some international standards require specific transfer encoding types. For example, [Japanese email](#) permits only base64 encoding in the message header fields.
2. The encoder negotiates with the restricted interface to determine what restrictions are in force. For historical interfaces like a UNIX pipe, this is often not possible; the restrictions are an intrinsic property of the interface and are not negotiable (or even visible) at runtime. Some interfaces, however, do provide a means to relax some of the restrictions. [Extended SMTP](#), for example, can allow 8-bit text if both the sender and receiver agree to it.
3. The encoder examines the data stream to determine what transfer encoding is necessary, if any. If the encoder only supports a single encoding type, or if the sender has explicitly specified an encoding type, then this step is omitted. If the data stream is large and buffering is not available, this step might be impractical.
4. The encoder checks the restrictions of the server against the characteristics of the source data stream, and chooses the most efficient transfer encoding type that still complies with the interface restrictions. The ideal case is an identity encoding.

5. The encoder writes its signal field to the restricted interface, indicating the type of encoding used. The signal might be a preamble string, like the `uuencode begin` line, or it might be part of a larger syntax like the MIME `Content-Transfer-Encoding` header field.
6. The encoder writes the encoded data stream to the restricted interface.
7. The restricted interface makes the encoded data stream available to the decoder.
8. The decoder opens the restricted interface and reads the signal field to identify the encoding type. The decoder must perform this check even if it only supports a single type. The decoder must be able to flag an error if, for example, the sender was configured to use `uuencode`, but the receiver was configured to use `bt oa`.
9. The decoder reads the encoded data from the restricted interface and verifies that the encoded data is correct. Depending on the protocol specification, the decoder can choose to ignore certain types of errors or try to reasonably recover from the error. For example, `uudecode` skips added ASCII spaces at the beginning of a line and can fill in missing spaces at the end of a line. If a decoding error occurs from which the decoder cannot recover, for example, the letter *G* in a hexadecimal value, the decoder must notify someone of the problem. Depending on the application, the decoder can return an error to the sender, notify the receiver, record the problem in a log file, or all of these.
10. The decoder writes the original data stream to the receiver.

Command Line Interface

All transfer encodings should be exposed through a command line interface, both for debugging and to facilitate their use in scripts. Historically, transfer encodings like `uuencode` and `bt oa` were implemented only in command line utilities. The notion of transfer encodings as a feature of a larger application, such as an email client or news reader, is relatively recent.

The encoder reads the source stream from the standard input or from a user-specified file. The decoder writes the original source stream to its standard output or to a user-specified file. The I/O interface must be "binary-clean" and able to read, buffer, and write all byte values, including all control characters and the NULL character. No special handling should be given to any characters. In C programming, this rules out all of the line-oriented `stdio` functions like `fgets(3S)` and `fputs(3S)`. Byte-oriented functions must be used instead, like `read(2)`, `fread(3S)`, `write(2)`, and `fwrite(2)`.

If the transfer encoding supports more than one encoding type, this must be exposed on the command line as an option with the most robust encoding type as a default.

Both the encoder and the decoder command line interfaces should be implemented as a small wrapper around an API. For more information, see ["Application Programming Interfaces."](#)

Character Interface

Not applicable.

Graphical Interface

Not applicable.

Application Protocols

Transfer encoding is an application protocol layer that is specifically designed to tunnel international text and binary data through storage and interchange interfaces that do not support 8-bit or binary data.

Storage and Interchange

A transfer encoding does not have any storage or interchange capabilities. However, transfer encoding does pose a design problem for the implementor of a storage and interchange interface. Consider the following scenario:

1. An email message is composed by a client application. The message contains a binary attachment.
2. The client delivers the message via SMTP to a mail server. Because SMTP does not support binary data, the client applies MIME `base64` transfer encoding to the attachment. The mail server inserts the message in a queue; that is, a storage interface.
3. At a later time, the mail server reads the message from the queue, and delivers it to a message store using a proprietary binary-clean interface.
4. An end-user retrieves the message from the message store using IMAP. Because IMAP does not support binary data, `base64` must be applied to the attachment.
5. The receiving client removes the `base64` transfer encoding and allows the user to save the attachment as a disk file.

While two of the interfaces in this scenario are restricted, most are binary-clean. This leaves the implementors of the mail server queue and the message store with a choice:

- Should they remove the transfer encoding when storing the message, and then reapply the transfer encoding when the message is retrieved?
- Should they just leave the message alone and store it encoded?
- Does the answer change if the message is routed using a binary-clean interface, like UNIX-to-UNIX Copy Protocol (UUCP)?
- Should there be a separate interface that can be called on demand to alter a message's transfer encodings just before delivery?

When MIME was first published and MIME-aware email clients were first available, all servers and storage interfaces left the transfer encoding untouched. This was done partly to minimize computational load on the servers, but mostly because developers could not justify the programming effort involved in managing the transfer encodings on the server.

As end-user demands for improved network performance and message fidelity have increased, however, servers have become increasingly sophisticated in their handling of transfer encoding. This trend will continue, based on the latest standards drafts.

Application Programming Interfaces (APIs)

Any implementation of a transfer encoding should be first exposed as an API, to facilitate the development of both command line tools and integration into large programs. A good API will offer the applications developer flexibility in when and how to apply encoding and decoding. For an explanation of the parameters that must be exposed to the API, see ["Transfer Encoding Data Flow."](#)

The API to a transfer encoding layer must be binary-clean and use buffer length parameters rather than NULL terminated strings.

When a higher-level protocol or service supports the use of transfer encodings, it is important that APIs which support that protocol or service, also support the transfer encodings. Many high-level API designers have omitted this step, moving the burden of supporting the transfer encodings onto the application developer. The interface must still expose flexibility to the applications developer; however, some APIs have so thoroughly buried the transfer encoding interface that the application is unable to obtain vital information or meet specific encoding requirements, such as those for [Japanese email](#).

Requirements for Compliance

Command Line Interface

A compliant transfer encoding interface guarantees that the binary data stream written by the decoder, that is, the consumer, is identical to that which was read by the encoder, that is, the provider. This is a straight-forward software quality assurance problem that lends itself well to automated testing. The test suite should:

- Check for the natural boundary conditions in the encoding type
- Include characters known to cause failures in the restricted interface
- Be first run with the encoder and decoder directly connected and then run with a restricted interface between them

A transfer encoding should not pass any parameters other than the encoding type from the encoder to the decoder. Passing text parameters is complex and better handled by the layer above the transfer encoding.

At least one known transfer encoding implementation does pass a text parameter that is supplied on the command line: `uuencode` passes a file name. This parameter is restricted to a 7-bit ASCII subset that varies from platform to platform, illustrating exactly why transfer encoding should not be overloaded with parameter passing.

Character Interface

No requirement.

Graphical Interface

No requirement.

Application Protocols

See the requirements under ["Command Line Interface."](#)

Storage and Interchange

See the requirements under ["Command Line Interface."](#)

Application Programming Interfaces

See the requirements under ["Command Line Interface."](#)

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

Sample Email Server Internationalization Assessment Matrix

Revision information

Matrix version: 1.0, April 2001

This checklist is in the form of a matrix that enables you to assess the internationalization status of a product. Across the x-axis are the categories known as interfaces. Down the y-axis are the categories known as objects and methods.

Each available box in the matrix should be populated with one of the following color-coded values:

- **Compliant** - Product fulfills all the requirements defined for this interface/object combination.
- **Partially compliant** - Product fulfills some of the requirements defined for this interface/object combination. An accompanying explanation in text form should appear after the matrix.
- **Non-compliant** - Product does not fulfill any of the requirements defined for this interface/object combination. Plans for future inclusion should be provided in a roadmap.
- **Not applicable** - Product does not provide the functionality of this interface/object combination.

		User Interfaces			Program Interfaces				
		Command line	Character	Graphical	Application Protocols	Storage and Interchange	Application Programming		
1. Translatable product components	1.1 Translation negotiations, defaults, and selection	Partially compliant (6)	Not applicable	Partially compliant (1)	Partially compliant (7)	Partially compliant (8)	Compliant		
	1.2 Textual objects	1.2.1 Fixed textual objects	Partially compliant (20)	Not applicable	Partially compliant (21)	Compliant	Compliant	Compliant	
		1.2.2 Messages	Compliant	Not applicable	Compliant	Partially compliant (9)	Partially compliant (10)	Partially compliant (11)	
		1.2.3 Help systems and documentation	Not applicable	Not applicable	Non-compliant	Not applicable	Partially compliant (22)	Not applicable	
	1.3 Non-textual objects	1.3.1 Icons, images, and colors	Not applicable	Not applicable	Non-compliant	Not applicable	Not applicable	Not applicable	
		1.3.2 GUI objects	Not applicable	Not applicable	Compliant	Not applicable	Not applicable	Not applicable	
		1.3.3 Sounds	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	
		1.3.4 Other	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	
	2. Cultural formatting and processing	2.1 Culture negotiations, defaults, and selection	Partially compliant (18)	Not applicable	Partially compliant (2)	Compliant	Compliant	Compliant	
		2.2 Abstract objects	2.2.1 Time, date, and calendar	Non-compliant	Not applicable	Partially compliant (3)	Compliant	Compliant	Compliant
			2.2.2 Numeric, monetary, and metric	Compliant	Not applicable	Compliant	Not applicable	Not applicable	Not applicable
			2.3.1 Ordered lists (collation)	Not applicable	Not applicable	Partially compliant (14)	Not applicable	Partially compliant (17)	Not applicable

	2.3 Structured text	2.3.2 Personal names, honorifics, and titles	Compliant	Not applicable	Compliant	Not applicable	Non-compliant	Not applicable
		2.3.3 Addresses	Compliant	Not applicable	Partially compliant (19)	Not applicable	Not applicable	Not applicable
		2.3.4 Other formatting and layout	Compliant	Not applicable	Compliant	Not applicable	Not applicable	Not applicable
	2.4 Language processing	2.4.1 Lexical and grammatical	Compliant	Not applicable	Partially compliant (15)	Partially compliant (12)	Not applicable	Not applicable
		2.4.2 Phonology and sound-to-text	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable	Not applicable
3. Text Foundation and Writing Systems	3.1 Writing system negotiations, defaults, and selection		Non-compliant	Not applicable	Partially compliant (4)	Compliant	Compliant	Compliant
	3.2 Plain text representation	3.2.1 Characters (semantics and codespaces)	Compliant	Not applicable	Compliant	Partially compliant (13)	Partially compliant (16)	Not applicable
		3.2.2 Strings (encoding methods and transcoding)	Compliant	Not applicable	Compliant	Not applicable	Compliant	Not applicable
	3.3 I/O and interchange	3.3.1 Transfer encoding (8-bit clean)	Not applicable	Not applicable	Not applicable	Compliant	Compliant	Not applicable
		3.3.2 Device input (keyboard and input methods)	Not applicable	Not applicable	Partially compliant (5)	Not applicable	Not applicable	Not applicable
		3.3.3 Device output (font management, rendering, and output methods)	Not applicable	Not applicable	Compliant	Not applicable	Not applicable	Not applicable

Explanations for partially compliant areas and notes:

1. Translation negotiations in Manager Console are limited by a combination URL, 110n, and file placement organization. It is somewhat complex for the user to choose a language based on the framework provided. Administrator is totally dependent on central Administrator negotiations.
2. Culture negotiations in Manager Console are limited by a combination URL, 110n, and file placement organization. It is somewhat complex for the user to choose a culture based on the framework provided. Administrator is totally dependent on central Administrator negotiations.
3. In order to accommodate calendars in Manager Console other than the Gregorian calendar, customer or 110n team must be able to extensively program in JavaScript. Also, time zone is not determined, GMT is used but not obvious to user.
4. Writing system negotiations in Manager Console are not handled properly. This is a limitation of the central Manager Console product. Administrator is totally dependent on central Administrator negotiations.
5. Some calls to the Input Method are being made, but implementation may be incomplete.

6. Message Store command line does not determine language of user. Admin command line goes by the server language. MTA command line is not compliant.
7. POP protocol does not allow for language negotiation. Authentication does not determine language. IMAP is compliant, as is MTA.
8. No language determination for logging (no viewer).
9. POP messages not yet externalized, but may be fixed by 5.0 release. MTA is limited by SMTP.
10. Log messages hard-coded - depending on whether we put a viewer mechanism in place at some point, this could become compliant.
11. Authentication log messages hard-coded.
12. IMAP search, missing some minor charsets.
13. Authentication allows UTF-8 passwords but not userids (not a requirement yet in the industry).
14. Administrator is totally dependent on central Directory sorting. Manager Console is non-compliant.
15. Administrator is totally dependent on central Directory search. Manager Console is non-compliant.
16. Character set conversions are weak in the Unicode area for the MTA. Message store is compliant.
17. MTA is constrained by standards, will offer a non-standard sort option in future release.
18. Admin CLI goes by the server time zone.
19. Cannot handle postal code before city or region name in the GUI.
20. Admin CLI is not compliant. Message store and MTA are compliant.
21. Manager Console is non-compliant. Administrator is compliant.
22. Online help is stored in a proprietary format - unknown whether this format can handle all charsets.

Important dependencies:

A. In many cases, the code is relying on Java to handle charset determination and conversion, locale determination, character parsing, and string parsing, without calling specific methods to accomplish each step.

B. Manager Console, Administrator, and Admin CLI rely heavily on the central Manager Console, Admin Console, and Database servers. The tasks these centralized servers are expected to perform are translation, culture, and writing system negotiations, charset conversions, search, and sort.

Location of product roadmap for providing future i18n functionality: None created.

{your product name} Internationalization Assessment Matrix

{Please change the <TITLE> and <AUTHOR> of this page when you use this template. You can then delete this line.}

Revision information

Matrix version: { **version, yy.mm.dd** }

Last assessment modification: { **yy.mm.dd, your name, your email address, what was updated** }

This checklist is in the form of a matrix that enables you to assess the internationalization status of a product. Across the x-axis are the categories known as interfaces. Down the y-axis are the categories known as objects and methods.

Each available box in the matrix should be populated with one of the following color-coded values:

- **Compliant** - Product fulfills all the requirements defined for this interface/object combination.
- **Partially compliant** - Product fulfills some of the requirements defined for this interface/object combination. An accompanying explanation in text form should appear after the matrix.
- **Non-compliant** - Product does not fulfill any of the requirements defined for this interface/object combination. Plans for future inclusion should be provided in a roadmap.
- **Not applicable** - Product does not provide the functionality of this interface/object combination.

		User Interfaces			Program Interfaces			
		Command line	Character	Graphical	Application Protocols	Storage and Interchange	Application Programming	
1. Translatable Product Components	1.1 Translation negotiations, defaults, and selection							
	1.2 Textual objects	1.2.1 Fixed textual objects						
		1.2.2 Messages						
		1.2.3 Help systems and documentation						
	1.3 Non-textual objects	1.3.1 Icons, images, and colors						
		1.3.2 GUI objects						
		1.3.3 Sounds						
		1.3.4 Other						
	2 Cultural formatting and processing	2.1 Culture negotiations, defaults, and selection						
		2.2 Abstract objects	2.2.1 Time, date and calendar					
2.2.2 Numeric, monetary, and metric								
		2.3.1 Ordered lists (collation)						

	2.3 Structured text	2.3.2 Personal names, honorifics, and titles						
		2.3.3 Addresses						
		2.3.4 Other formatting and layout						
	2.4 Language processing	2.4.1 Lexical and grammatical						
		2.4.2 Phonology and sound-to-text						
3. Text foundation and writing systems	3.1 Writing system negotiations, defaults, and selection							
	3.2 Plain text representation	3.2.1 Characters (semantics and codespaces)						
		3.2.2 Strings (encoding methods and transcoding)						
	3.3 I/O and interchange	3.3.1 Transfer encoding (8-bit clean)						
		3.3.2 Device input (keyboard and input methods)						
		3.3.3 Device output (font management, rendering and output methods)						

Explanations for partially compliant areas and notes:

Important dependencies:

Location of product roadmap for providing future i18n functionality:

Chapter 2 Planning for Internationalization

2.1 Planning Ahead

Products cannot be internationalized in a day or even in the course of one release cycle. Internationalization affects all stages of software development, from identifying requirements to production. Since internationalization might be implemented over a period of several release cycles, a plan or "road map" to internationalization needs to be created.

This chapter helps identify priorities for internationalizing a product over time. Along with the completion of the matrix, this information can form a long-term road map to full internationalization. Road maps can apply to existing products as well as new ones.

In order to determine the route to full internationalization, product users must be identified and components must be evaluated. The next two sections describe different types of users and components.

2.2 User Categories

Products provide interfaces and services to a variety of users. Users can be divided into two categories:

- **Internal users** - Product designers, developers, testers, technical support personnel, sales engineers, consultants, and all who work on product creation, release, support, installation, and customization.
- **External users** - People who use the product and who are not internal users.

Obviously, a product can have many more categories of users, but from an internationalization perspective, these are the primary ones. To prioritize the internationalization steps, external users can be further subdivided into the following categories:

- **Non-technical users** - People who are not required to be technically savvy to use the product. Non-technical users are the highest priority for full internationalization. A non-technical user might be someone using an email client, Web browser, or word processor. Typically the product is client software.
- **Technical users** - People who require some technical expertise to use the product. Technical users are the second highest priority for full internationalization. Someone using a programming language, debugging tool, network monitor, or software design tool can be considered a technical user. In this case, the product is often client software but can also be server software.
- **Technical administrators** - People who install, configure, and maintain the product. Technical administrators are the lowest priority for full internationalization. Technical

administrators usually install and configure server software using command line options, configuration files, and administration consoles to monitor the successful execution and stability of a product.

The relationship between user categories and internationalization planning is further discussed in [section 2.4](#).

2.3 Data Categories

Software products are designed to perform some sort of function, usually data processing. Users request the function using an interface that consists of messages, windows, buttons, form fields, drop down lists and other screen elements.

For internationalization, software data can be broken down into:

- **Functionality** - What the product does. This is the highest priority for internationalization.
- **Messages** - Text based information from the product for the user. This is the second highest priority for internationalization.
- **Interface elements** - Text containers and graphics, such as windows, buttons, and icons. This is the lowest priority for internationalization.

Functionality is best internationalized by planning it from the start. Retrofitting internationalization into existing functionality is like changing the foundation on an existing house--the entire structure is affected. Adding internationalization functionality at the end is an expensive and frustrating exercise and often requires a rewrite of most of the code.

Messages, due to their static nature, are fairly straightforward to internationalize. First, they must be further divided into internal and external messages. Internal messages are often called debug messages and are intended for internal users only ([see section 2.2](#)). They can be activated by some sort of switch, but this switch should not be in the external documentation. People who need to debug, support, or maintain the program code of the product can turn on internal messages.

External messages can be seen by any user. They can also be activated by switches, but these switches are documented. Some examples are: status messages, error messages, exception messages, and help messages. If an external user can activate the message, it is an external message.

Internal messages must **not** be internationalized. This prevents them from being localized accidentally, avoiding difficulties for support and customization.

Interface element internationalization is slightly more complex than that of messages. Planning the interface usually requires an interface designer who is aware of internationalization issues. To internationalize some interface elements, it is sufficient to put them into resource files; however, for other interface elements, such as icons, this might not be enough. Designing a new icon for each locale is extremely expensive; therefore, it is not usually done. The icons provided with the base interface are used all over the world. This means icons should be considered in the requirements and design stages of the development

cycle.

Functionality, external messages, and interface elements should all be internationalized. Only internal messages are not internationalized. More planning information for data categories is provided in the next section.

2.4 Designing the Road Map

In designing an internationalization road map, priorities must be established. The following considerations are listed in order of priority:

1. A user perceives two aspects of a software product: the user interface and the functionality. While a simple and attractive user interface can help sell a product, it cannot make up for incorrect functionality. People buy a software product for the functions it performs; otherwise, it is useless to them. Functionality, therefore, is a higher priority than messages and interface elements.
2. Messages are the means by which the software communicates with the user. If something goes wrong, or the user's request is incomplete, it is through messages that the user understands what to do next. Without messages in a familiar language, the user can become frustrated and telephone a support center. This is costly both in low customer satisfaction and in support costs; therefore messages become the next priority.
3. Interface elements have the lowest priority of the three data categories. This is not to discount the importance of interface elements; they are the product's look and feel and can make product use much easier.
4. Non-technical users are usually far greater in number than technical users or administrators. In addition, non-technical users tend to be the least likely to be familiar with or use English computer-related terms. As a result, product internationalization is most exposed to this group. Their functionality, messages, and interface elements should be the first areas to be internationalized.
5. Technical users may understand the terminology within their area of expertise, but that usually represents a small portion of the data in a software product. For software development tools, internationalization functionality can be crucial to the success of the product. Other companies want to create internationalized software as well. Technical users are a higher priority than technical administrators.
6. Last but not least are technical administrators. Administrators are people too. While many administrators can manage with English data, they would function much more efficiently if data were available in their language.

While many of these considerations can only be addressed through localization, that localization cannot be accomplished without first internationalizing the product. Localization decisions should not be restricted due to lack of internationalization.

2.5 Additional Considerations

There are more considerations than those listed in the previous section. While internationalization strives for a single generic code base, the reality is that some languages need custom modules for support. For example, in a word processing product, extra rendering functionality is required to handle Arabic. This type of functionality is not likely to be part of the general code base, though it should be easily plugged into the general code and should blend seamlessly into the application programming interface (API). In designing the road map for this product, markets should be analyzed.

If a product is a provider of internationalization functionality ([see Chapter 1](#)), it is more important to include as much internationalization as possible. Providers should also have a clear road map. Consumers depend on their providers for a certain amount of internationalization and their road maps follow those of their providers.

This chapter aims to provide general guidelines for internationalization phase planning and the underlying reasoning behind it. The [Internationalization Assessment Matrix](#) helps you to break down areas of internationalization even further. Product managers and engineers should take into account priorities relating specifically to their products.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.

Chapter 6 Reading the Matrix

6.1 Audience

The matrix is technically oriented and can seem daunting to people who are not involved in technical product development. This chapter is intended for people who need to read completed matrices and determine product status. This includes people making sales and localization decisions, high level managers, product marketers, and others in similar positions.

This chapter is not intended as a shortcut to filling out the matrix. To understand the fields in the matrix, you must read the appropriate section.

6.2 The Matrix at a Glance

Status entries in the completed matrices should be color-coded. This is to aid in determining overall product status at a glance. The following legend should be at the top the completed matrix:

- **Compliant** - Product fulfills all the requirements defined for this interface/object combination.
- **Partially compliant** - Product fulfills some of the requirements defined for this interface/object combination. An accompanying explanation in text form should appear after the matrix.
- **Non-compliant** - Product does not fulfill any of the requirements defined for this interface/object combination. Plans for future inclusion should be provided in a roadmap.
- **Not applicable** - Product does not provide the functionality of this interface/object combination.

A fully internationalized product has a black and green matrix. In the majority of cases, purple is interspersed throughout. To determine the severity, see "Explanations for Partially Compliant Areas and Notes" at the bottom of the matrix. If there are several red entries, the product requires a lot of internationalization (i18n) work.

6.3 What do the Boxes Represent?

The following categories are displayed across the top of the matrix:

- **User Interfaces** - Covers the product areas that are displayed to the user and that involve user interaction. This includes all textual and non-textual elements, such as dialog boxes, error messages, and command line help.

- **Program Interfaces** - Although program interfaces are buried within the code and are not directly visible to users, they support user-visible components by passing data correctly, providing appropriate headers and labels and converting formats.

The following categories are displayed down the left side of the matrix:

- **Translatable product components** - Describes product items that you can translate into another human language. I18n is clearly visible as these items can be translated or localized.
- **Cultural formatting and processing** - Describes product items that have different formats in different cultures. This differs from localization in that the formatting occurs based on user locale and not localization. For example, in the U.S., dates are displayed in month/day/year format. In the U.K., dates are displayed in day/month/year. The English product may be used in both regions, but the date format needs to change. Product functionality may also require i18n, depending on what the product does.
- **Text foundation and writing systems** - Describes text handling. This mainly involves a programmatically controlled area and focuses on the i18n of functionality that relates to text processing.

6.4 Relating the Matrix to the Product

To understand the matrix, it is best to know what the product is used for and who is likely to use it. Define what sort of data the product receives as input and what it is expected to do to this data. With this information in mind, examine the matrix. The following scenario illustrates how you might use the matrix to check your product's i18n requirements.

Scenario: I18n Verification

Susan Smith, a globalization director, is trying to determine what countries to sell an email server in, and what languages to localize the interface into. She has received [this matrix](#) (open this sample matrix in a new window for side by side viewing).

She examines the Translatable Product Components category row by row:

- **Translation Negotiations, Defaults and Selection** - This row describes the process of determining what language to serve to the user. Since Susan is concerned about localizing the product, she notes that several interfaces are only partially compliant. Reading notes 1, 6, 7, and 8, she finds that:
 - Language selection for the Manager Console may need to be well-documented for the customer
 - It is unknown whether the Administrator will handle translations, since there is no i18n information on the central product
 - Command line is unlikely to handle localized messages very well

- Login messages may not be translatable
- Logs must be in English

- **Fixed Textual Objects** - In this row, the issue is whether translation will break the functionality; fixed text is untranslatable text, such as keywords. The fact that there are two **partially compliant** boxes suggests that the translators may need additional notes to avoid translating text which is fixed, or the translated product may require several cycles of testing, fixing, and rebuilding to make sure that functionality has not been affected by the translation.

- **Messages** - This row represents translatable text. There are three notes, 9-11, which basically state that the log messages are not translatable.

- **Help Systems and Documentation** - Help and documentation may be complex to translate and there is a chance that help may not work in some languages; this is described in the next row.

- **Icons, Images and Colors** - The icons and other images are not culturally suitable, or they may contain lots of imbedded text. Someone will need to review them to determine whether they are appropriate for any international markets and how much work would be involved in creating new icons.

- **GUI objects** - These objects are compliant, which means all the screens have flexible components.

- **Sounds** - Not applicable.

- **Other** - Not applicable.

Susan then moves on to the Cultural Formatting and Processing category. In the Culture Negotiations, Defaults, and Selection row, Susan is interested in the ability of the program to determine the locale of the user. Note 2 suggests that the locale may only be communicated in a localized product, at least for Manager Console users. Note 18 is not a concern for her, since Admin CLI is server admin.

The next eight rows, sections 2.2.1 through 2.4.2, are concerned with handling and formatting data according to the user locale. Looking at the Time, Date and Calendar row, she sees that the command line cannot handle locale-specific formats, and that the GUI is only partially compliant. Sorting is not handled according to locale in some cases, and the storage of personal names may not handle different name formats. There is a problem in the GUI where postal codes which precede the city or region cannot be accommodated. Searching is not consistent throughout the product.

Finally, Susan examines the Text (Writing System) Foundation category. All user text must be in a character encoding of some kind. In order to correctly parse the data, the program needs to know the character encoding or charset. From this row, she realizes that the

command line cannot determine the charset; at a minimum the expected charset should be documented. It also looks like text input using the Manager Console is at risk.

She is pleased with the character handling; the partially compliant notes discuss the limitations of an existing standard, and the need for the addition of a few converters.

Strings are completely taken care of, as are the transfer encodings. This means that words and phrases are processed and stored correctly without corruption, and 8-bit data can pass through the system without a problem.

Device input has a minor issue, an unknown functionality with input method editors (IME). IMEs are used for typing complex writing systems, such as Japanese. It looks like this will have to be tested to verify consistency.

Device output has no problems, so Susan knows that text will display on the screen properly. Now it is time for her to take this information and compare it to the data she has on the international markets. She must determine whether it will be cost effective for her to launch a campaign for the English product in some places and localize the product for others.

6.5 Potential Showstoppers

It is important to remember that no two products are alike, and the importance of any square in the matrix is relative to the product itself. Sales and localization decisions must be based on individual product features and i18n support for each feature. However, there are several boxes in the matrix, which if marked **non-compliant**, could indicate serious limitations for international customers. This section identifies areas that require careful consideration:

- **1.1 Translation negotiations, defaults and selection** - From a localization perspective, this section is key if the product is expected to handle several translations as part of a single installation. If the user's preferred language is not determined by the program, it cannot display the correct language interface.
- **1.2.2 Messages** - This section is even more critical for localization. If messages are non-compliant, they are not localizable.
- **1.3.2 GUI objects** - If this section is non-compliant, translations might be unreadable.
- **2.1 Culture negotiations, defaults, and selection** - If this section is non-compliant, then it is impossible to deliver data that is formatted in a culturally appropriate manner. This means that dates, numbers, currency figures, addresses, sorting, and similar objects cannot be processed or displayed in a format expected by users in different countries around the world.
- **2.2.1 Time, date, and calendar** - A calendar program is of no use if it cannot display the regional calendar and date formats that apply to the user's locale.

- **2.3.3 Addresses** - If the product is an address book, for example, and this section is non-compliant, there is no point in selling the product outside the region for which it is designed.
- **3.1 Writing system negotiations, defaults and selection** - If the program cannot determine what writing system the data is in, it cannot parse it correctly. This may not be critical if the product only accepts limited data as input, for example, a calculator program which recognizes 0-9 and a few simple mathematical symbols. Most products, however, handle text of some kind and so non-compliance can block international distribution.
- **3.2.1 Characters (semantics and codespaces) and 3.2.2 Strings** - These sections are critical for text processing. If either of them are non-compliant, the result may be corrupted data.

Copyright 2001 Sun Microsystems, Inc., 901 San Antonio Road, Palo Alto, CA 94303 USA. All rights reserved.