INSIDE MACINTOSH

Text Handling and Internationalization

WWDC Release

May 1996

Apple Computer, Inc.

Apple Computer, Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

The quotation on page 35 is by Emily Dickinson.

The quotation on page 37 is by Lao Tsu.

The quotations on pages 26 and 30 are by Oscar Wilde.

Contents

Introduction to Text Handling and Internationalization on Mac OS 8

Contents

This chapter provides an overview of text-handling and internationalization support in Mac™ OS 8. For the Mac OS 8 Developer Release: Compatibility Edition, this chapter is meant to serve these purposes:

- It provides a high-level view of the new and improved text handling and internationalization components on Mac OS 8, describing them and highlighting their special features. See "About Text Handling and Internationalization in Mac OS 8" (page 1-6).
- It explains System 7 backward compatibility provided within Mac OS 8 and identifies those few areas for which backward compatibility is not provided. See "Mac OS 8 Exceptions to Backward Compatibility With System 7" (page 1-11).
- It gives a brief look at the future vision for text handling and internationalization beyond Mac OS 8. See "Looking Toward the Future" (page 1-13).
- It explains the fundamental conceptual differences between Mac OS 8 and System 7 that provide the underpinnings to text handling. This includes explanation of a different understanding of the concepts of a writing system and a script for Mac OS 8 from that on which text handling for System 7 was premised. The Mac OS 8 view is more aligned with the concepts of scripts and writing systems as they are used and understood in the area of linguistics. However, it requires an adjustment in thinking on your part if you have based your understanding of these concepts on how they are explained in *Inside Macintosh: Text* for System 7 in relation to System 7's Script Manager. See "Text Handling in System 7 and Mac OS 8: A Comparison" (page 1-14).
- It gives a more thorough treatment of several of the new text components central to Mac OS 8 that make internationalizing your application easier. It describes various aspects of these three components:
  - Text objects and the Text Object Manager. See "Text Objects for Text Storage and Interchange" (page 1-25).
  - Locales, the locale database, and the Locale Object Manager. See "Storage and Retrieval of International Data and Preferences" (page 1-48).
  - Encoding conversion, the Low-Level Encoding Converter, and the High-Level Encoding Converter. See "Text Encoding and Conversion" (page 1-67).

- It briefly describes the Text Editing Services and the String Comparison Services. For Text Editing Services, see "Handling Editable Text" (page 1-84). For the String Comparison Services, see "String Comparison" (page 1-89).

▲ **WARNING**
This document is preliminary and incomplete. All information presented here is subject to change in later developer releases. Some information it contains will become the basis for conceptual and tutorial information in chapters of the *Inside Macintosh: Text Handling and Internationalization* book for Mac OS 8, to be provided at a later date. ▲

About Text Handling and Internationalization in Mac OS 8

Mac OS 8, provides text-handling and internationalization features that carry forward the Apple® tradition of setting new standards and leading the industry in software internationalization. By building into its design ease of use and powerful flexibility, Mac OS 8 gives developers more control over how an application can present language-based choices to end users and provides extensibility that can move with any direction the industry takes. Mac OS 8's international software allows you to develop world-ready software that can be released in more than a single geographic market at the same time.

One example of the flexibility inherent in Mac OS 8 is that it allows you to create an internationalized application that can handle a mix of any text encodings (or text encoding schemes) and be easily localized for any language and geographical region. In addition to the standard set of Mac OS 8 text encodings and Unicode shipped with the system software, you as a third-party developer can provide your own text encodings and make them available to applications running on Mac OS 8. (For System 7 and its earlier versions, this was not possible.) For example, any of the DOS code pages can be installed on a Mac OS 8 system. Mac OS 8 supplies a number of text encodings including Latin-1 (ISO 8859-1, which is the default encoding for the Internet. Mac OS 8 performs conversion from one encoding scheme to another. One benefit is that your application can support text files in any encoding scheme that your user might obtain from the Internet.

Mac OS 8 provides more flexibility in its international support than other platforms do. Neither Windows 95 nor Windows NT (New Technology) offer feasible alternatives to Mac OS 8.

Mac OS 8 includes far more Unicode support than does Windows 95 and it offers far more flexibility and ease of use than does Windows NT. For example, if you code your application to Windows NT using its standard form, you can use the Unicode text encoding or another text encoding, but not both. Windows NT standard form does not let you support a mix of text encodings; the single encoding to be used is set when you compile your application. You can, however, make specific calls from within your application to support different text encodings, but this approach is far more cumbersome than the easy way in which Mac OS 8 allows you to support mixed encodings.

Mac OS 8 Text-Handling Component Features

Mac OS 8 provides many new text-handling components. The features these components offer include

- support for text objects that let you store encoding specification, language, and region information along with text. Because they encapsulate this information, text objects remove the complexity from the work you need to do to maintain the text encoding for text along with the text string. Text objects are the primary means of passing text to and between system components. Text objects allow easy conversion of text between encoding schemes and easy localization of your application; you should use them for text displayed as part of the human interface, such as text shown in menus and dialog boxes. The **Text Object Manager** provides support for text annotations that let you attach related data to a segment of a text string within a text object. For Mac OS 8 applications, you should think of using text objects as the default scenario for handling text. The only circumstances in which you might not want to use text objects are when you export text to another application on another platform and when you implement a text-intensive application such as a word processor. See “Text Objects for Text Storage and Interchange” (page 1-25) for more information.
- support for use of a new repository for international preferences and data, called the locale database, and access to the database and its contents through the **Locale Object Manager**. You can add objects containing data to the database and remove them from it, search the database for objects, and obtain information about objects. See “Storage and Retrieval of International Data and Preferences” (page 1-48).

- the provision of a full-scale encoding converter that allows you to convert text to and from Unicode, gives you fine-grain control over the conversion process, and provides extensive error reporting.The **Low-Level Encoding Converter** supports table lookup-based conversion to or from Unicode. It also provides attendant utilities, such as truncation functions and functions for converting Pascal strings. See "Text Encoding and Conversion" (page 1-67) for more information.
- the provision of a high-level encoding converter that allows you to convert text between any two encodings or schemes and offers ease of use by determining default conversion-process values for you. The **High-Level Encoding Converter Manager** performs table lookup-based and algorithmic conversions. It uses the Low-Level Encoding Converter for table lookup-based conversion and plug ins for algorithmic conversions. This version of the converter does not map external formatting from the source text to the converted text, so it is best used to convert mainly plain text or text with inline formatting, such as HyperText Markup Language (HTML). You might want to use either of the encoding converters instead of text objects when your application does extensive text processing, in which case you'll need to perform encoding conversions yourself. See "Text Encoding and Conversion" (page 1-67) for more information.
- **Text Editing Services**, including a text panel, text engines, and an enhanced and improved version of **TextEdit**. Text panels are simple to use, requiring very little effort on the part of your application. They allow you to display editable text fields in your application's windows. The text panel manages itself in the rectangle you define. You can select the text engine to use with a text panel. For more extensive processing, you can use an engine alone. For Mac OS 8, you are not limited to use of a single text engine as is the case with TextEdit in System 7. Text Editing Services include an enhanced version of TextEdit that provides support for integrated inline input and text objects, and is based in the new event model. See "Handling Editable Text" (page 1-84). Note that for this release, a modified version of the TextEdit engine that eliminates the 32K record limitation is the only supported text engine.
- new and enhanced **String Comparison Services** for comparing and searching strings for all languages. These functions support text objects, allowing two strings in different text encodings to be compared. See "String Comparison" (page 1-89) for more information.
- an enhanced version of the **Text Services Manager** (TSM) that includes support for a broad range of text services in addition to input methods,

removes Chinese-Japanese-Korean (CJK) limitations on input-method support, and greatly simplifies the process required to make your application TSM aware. TSM takes full advantage of the new event model. For Mac OS 8, TSM supports new categories of text services through a common interface and makes these services available to a wide range of applications. To enable integration of these services within your application, TSM supplies these two levels of interfaces:

- a high-level interface through functions that use the `TSMDocumentID` structure and that gives your application easy access to input methods and available interactive text services
- a low-level interface that manipulates the underlying `TSMContext` service context and allows your application direct access to the service's functionality, letting you use any services as an integrated part of your application.

If you are providing text services, TSM for the Mac OS 8 is designed to make it easy for you to create them. TSM supports these three categories of text services:

- input methods, designed to filter events passed to the application. Input methods are capable of intercepting text entry and interacting with both your application and its user to convert raw events to the text stream.
- interactive text services such as spell checkers, style checkers, and dictionaries, which, when activated, interact with your application and its user to perform a specific action on the text encapsulated within the application. They use the TSM protocol to access and modify the encapsulated text and interact with the user on behalf of your host application.
- batch services such as hyphenators, tokenizers, and stemmers that your application can use to obtain specific linguist processing of the text it handles. These services do not process events, and your host application must call them directly.

- revised and extended keyboard-menu handling support extended to handle other types of text input. The new design allows for additional classes of text input methods to be added to the menu and expands the technology to handle input devices other than the keyboard, such as speech-to-text and pen input. This support is now referred to as **Text Input Menu Handling**. The new menu, which is called the Text Input menu, can be configured by the application or its user. Your application can enable and disable menu items. For example, if it doesn't handle Arabic, your application can gray out the item.

- a new event model that includes a suite of **Text Events**, and a newly designed keyboard architecture with a new key translation module and functions. The **Key Translation** module is responsible for converting virtual keycodes into renderable character codes. New functionality is added to the translation routines for sequential dead keys and for mapping a single keypress to many character codes.
- a friendlier and simplified **International Text String Parser** that provides an interface composed of multiple functions and data structures rather than one function with a huge parameter block, as was the case in the past with single `IntlTokenize` function that used the `'itl4'` tokens resource. This set of utilities allows you to define your own metaclasses of tokens.
- **Number Formatting and Conversion Services** that give you the ability to format numbers for any language, country, and encoding scheme in a transparent way and that include scanning routines to convert the text into binary representation of the number.
- new **Date-and-Time** support through the use of TimeObjects and calendars. TimeObjects provides UTC (Coordinated Universal Time) support and an expanded range of representable times. The new Date-and-Time support also includes extended formatting with TimeObjects. A date-time format is defined for specifying information necessary to represent a TimeObject, or portions of one, in textual form. The calendar services use TimeObjects to provide a higher level of date and time support. These services provide a plug-in architecture that allows new calendar engines to be added and allows for the behavior of existing calendars to be overridden. Mac OS 8 provides support for a basic set of calendars. Calendars not supported in the first release of Mac OS 8 can easily be added later by Apple or, by you, as third-party developer.
- a new **Language Manager** that allows users to interact with the system in the language of their choice. The language an application uses need not be the same one that another application or workspace (such as the Finder) is using at the same time. At application launch, the Language Manager establishes the primary language for the application.
- additional **Text Utilities**.
- integration of WorldScript® I and WorldScript II supporting a single code base. This integration allows for one system for the world; there are no extensions and no patches.

A Word About Text Imaging in Mac OS 8

Mac OS 8 differs from System 7 in how it provides support for and handles text imaging. Apart from high-level text imaging done through text panels, text engines, TextEdit, and text objects, for Mac OS 8 text imaging is separate from text handling. Applications requiring greater control over text imaging and more intensive text-imaging services can use one of the graphics systems provided with Mac OS 8, such as GX or Color QuickDraw™.

The Text Object Manager provides default text imaging. A single text-imaging function combines the work of the QuickDraw Text text-measuring function and the Font Manager font metrics function, providing the width in pixels of the text object's string as imaged by `DrawText`, along with the total line height and the ascent. The Text Object Manager uses a color graphics port that you can specify. It also allows you to provide it with a font-substitution hint to assist it in determining which font to use when the most appropriate one is not available.

Mac OS 8 Exceptions to Backward Compatibility With System 7

With few exceptions, software you develop today using the international technology provided by System 7 will work with Mac OS 8.

IMPORTANT

The following list might be incomplete. More information will be provided in later developer releases. ▲

Here are aspects of the features of System 7 for which backward compatibility is not supported:

- Input methods. If you provide an input method for System 7 and want to make it available for Mac OS 8, you must replace it with a new one that is SOM-based (System Object Module). In System 7, input methods are implemented as components. In Mac OS 8, input methods are SOM-based.
- Aspects of the Script Manager.
  - The Script Manager's internal data structures are different in Mac OS 8 from what they were in System 7. If you access them directly in your

application—a behavior that is unsupported and was always unsupported in System 7—your application will not run in Mac OS 8.

- For most selectors, the `SetScriptVariable` and `SetScriptManagerVariable` functions have only local effect in Mac OS 8. That is, any changes you make using these functions will be effective only in your calling application's current context.
- When possible, the Script Manager's notion of a system script will be carried out and synchronized with the application's initial locale—usually the workspace locale. However, circumstances can occur in which an application's default locale has no equivalent script code.
- The functions `GetScriptUtilityAddress` and `SetScriptUtilityAddress` are no longer supported.
- For Mac OS 8, default fonts are specified by a special data structure in the locale. Mac OS 8 does not recognize equivalent values stored in the System 7 international resource.

- Aspects of QuickDraw Text.
  - You should avoid using the System 7 `FontToScript` and `FontScript` functions, which convert a family FOND ID to a script code. Although existing fonts retain their IDs for backward compatibility, new fonts are not backward compatible.
  - The Print Action routine, (described in System 7's *Inside Macintosh: Devices*) has no effect in Mac OS 8.
  - The `smCharPortion` verb is not supported.
  - The `ForceFont` flag is always false in Mac OS 8.
  - Negative verbs for `GetScriptVariable` that returned vectors for low-level imaging within WorldScript I will return universal procedure pointers (UPPs) to no-operational (no-op) routines for this release of Mac OS 8. This functionality will be supported in a later release of Mac OS 8.
- Aspects of keyboard support.
  - Although backward compatibility is provided for the `KeyScript` routine, Apple strongly recommends that you move to using the new Text Input Menu Handling support.
  - The `GetScriptManagerVariable` function called with `smKCHRCache` verb will return a KHCR, but not necessarily the one used by the Key Translation Manager.

- Aspects of WorldScript.
  - WorldScript I QuickDraw patch. Mac OS 8 does not support the `GetScriptQDPatchAddress` and `SetScriptQDPatchAddress` functions used for getting a pointer to the specified WorldScript I QuickDraw patch for a script system and patching a script system with a new QuickDraw routine, nor does it support the `GetScriptUtilityAddress` and `SetScriptUtilityAddress` functions.
  - For this release of Mac OS 8, none of the routines in the `WorldScript.h` header file are implemented. It is highly likely that they will not be supported at all for Mac OS 8. Later developer releases will provide further information.
- Dictionary Manager. The System 7 Dictionary Manager is no longer supported.
- Aspects of TextEdit.
  - Private scrap handling. For System 7, monostyled TextEdit used the private scrap in some circumstances. For Mac OS 8, monostyled TextEdit scrap handling is unified with multistyled TextEdit, and it always uses the public scrap.
  - Undocumented low-memory globals are no longer maintained. They include `TEFindLine`, `TETrimMeasure`, `WordRedraw`, `TEWdBreak`, `JPixel2Char`, `JChar2Pixel`, and `JHiliteText`.
  - For System 7, TextEdit calls the Script Manager to compute word breaks. The recommended way to customize this for Mac OS 8 is to use `TEDoText`.
  - If your application supports TSMTE, you should now get all of your text, not just two-byte text, through callbacks instead of through key events.

Looking Toward the Future

One of the principal underlying design goals of Mac OS 8 text-handling and international support is to provide extensibility not only within the current version of the system software but also with a view toward the future. This goal is reached in many areas. For example, text objects allow movement toward a system based in Unicode, if that direction is taken. Also, the design of the Locale database allows for storage and retrieval of any type of international data; as new requirements emerge, the database can easily accommodate them.

Support for Unicode is another example of this extensibility and open orientation. In addition to the fact that it offers the simplest solution for fully multilingual systems, Unicode offers many important and useful features. One of its most important aspects is that it naturally lends itself to text interchange among different platforms, as well as among applications and platform code on a single system.

Apple recognizes that the movement toward universal use of Unicode might not happen immediately or entirely; the industry might take another direction. For this reason, Apple intends to support and handle as many text encodings (and text encoding schemes) and the coded character sets they include as possible. Mac OS 8 is not limited to handling just a few encoding schemes based on the current market size. Apple recognizes that Unicode will not suddenly replace all other text encodings and that most platforms will have to deal with a mixture of other text encodings (and text encoding schemes) in addition to Unicode. For a description of the terms text encoding, text encoding scheme, and coded character set, see "Characters, Codes, Text Encodings, Text Encoding Schemes, and Text Elements" (page 1-70).

Because Mac OS 8 supports any encodings, including Unicode, it can move in any direction. If the industry moves toward Unicode, Mac OS provides support for it; if it doesn't, Mac OS 8 still supports conversion among any encodings and encoding schemes.

Not all of Mac OS 8 system components are Unicode based. Mac OS system software transition to Unicode most likely will occur gradually with different system components moving to Unicode at different times. By using text objects in applications you are porting or coding to Mac OS 8, you are ensuring that changes you will have to make to your code are few, if any, if Mac OS 8 transitions from a system that provides international support based in multiple encodings to one that supports a single encoding, Unicode.

Text Handling in System 7 and Mac OS 8: A Comparison

Mac OS 8 text-handling support shapes the context in which the future of internationalization is beginning to emerge by addressing software engineering requirements for developing applications for the global marketplace in ways that offer ease of use, extensibility, and flexibility.

Mac OS 8 provides support for more areas of text handling and internationalization than did System 7 and provides new solutions to problems addressed by System 7. This new support and these solutions give you more flexibility and control over how you perform text handling in your internationalized application and how you present language-based choices to your application's user. These solutions, implemented through the use of text objects, encoding conversion, and the locale database, and effective through your application's use of the Text Services Manager, text panels, String Comparison Services, and other Mac OS 8 text-handling components, allow you to build internationalized applications that require far less management in your code than was needed in System 7.

This section looks at the advantages of designing and developing internationalized applications and describes some of the conceptual and behavioral differences between Mac OS 8 and System 7 related to multilingual support.

Internationalization and Localization

Users of computers interact with them through a combination of elements implemented in system and application software that includes images and the written language. Text handling is heavily culturally dependent, and there are more software engineering issues associated with it than there are with the handling of images across languages and cultures. Among the cultural differences reflected in the written language are how the language is represented—for example, is it alphabetic, ideographic, or syllabic—and how national conventions for the presentation of date, time, and numbers are defined.

Note
Although the term ideographic is commonly used to characterize languages (such as Chinese) that include ideographs and pictographs, the term is inaccurate and misleading. Most so-called ideographic scripts include some ideographs and pictographs but they also include phonetics. For example, while Chinese includes both ideographs and pictographs, it also includes many complex characters that are phonetically based. ◆

In designing software applications that address these cultural differences, applications developers can follow one of these two general strategies:

- **Localization by reengineering**. You can design and develop an application that is specialized for a single language and culture from the beginning. Applications localized by reengineering generally include hard-coded dependencies on cultural and linguistic conventions. After you develop the initial specialized product, localization by reengineering entails adapting the software to fit specific national languages and cultural conventions for markets other than the one for which you originally designed it.
- **Internationalization**. You can develop application software that is generalized and designed to accommodate various languages and cultures. The process of designing and creating software with multiple cultures in mind—software that can be easily localized for various geographical regions and their languages without requiring changes to the source code—is called **internationalization**. This process entails distinguishing cultural elements that the software must accommodate differently for each language when the software is localized and handling those elements in a way that allows for variation. Internationalized software inevitably entails localization. Internationalized software can call functions that access and obtain data at runtime that is specific to a language or culture. Localizing software designed for an international market usually requires changes to the data or text of your application's user interface, but no source code changes.

Clearly, internationalization offers the more cost-effective and efficient design strategy if your intention is to make software meant for the global marketplace. Building localized software from the beginning requires multiple code bases if you plan to market your product in multiple geographical areas. Building internationalized software from the beginning allows you to develop and maintain one code base which localizers—developers in various countries who adapt software to those countries— can then specialize.

In considering what constitutes an internationalized application, these two separate issues surface:

- **User-interface handling**. It is important for the user interface portion of your application—that is, read-only text such as menu contents and system messages—to be easily localizable. (Mac OS 8 contains multiple localizations for system software so the system component can present messages in your user's language of choice.) You can use text object resources for the nonmodifiable data of your application's user-interface to allow for easy localization.
- **User input and editable-text handling**. It is important that the content portion of your application—that is, the part that deals with user input and

editable text—be able to handle text in any language. In other words, your application should allow your user to enter and edit text in any language. A single document might contain text in more than one language, so your application should support a mix of languages within the same document. A user might type text in German, then switch to Japanese. Mac OS 8 text handling components enable you to implement these processes easily.

To facilitate localization of applications, Mac OS 8 internationalization components provide the ability to store and access data required for specific locales. By supporting any text encodings and text encoding schemes, Mac OS 8 provides encoding conversion automatically through text objects or directly through one of the encoding conversion managers. In addition to the standard Mac OS 8 encodings provided with the system, Mac OS 8 supports Unicode and text encodings and text encoding schemes provided by third-party developers.

Note
A text encoding usually contains the encodings for the characters belonging to a single character set addressing a single script. A text encoding scheme is a method that allows for the support of and addresses multiple coded character sets.Text encoding schemes often include predefined escape sequences that indicate transitions to specific coded character sets. For a more complete description of the terms text encoding, text encoding scheme, and coded character set, see "Characters, Codes, Text Encodings, Text Encoding Schemes, and Text Elements" (page 1-70). ◆

Writing Systems and Scripts

Writing systems and scripts are viewed and understood differently in Mac OS 8 from System 7. Mac OS 8 text handling and internationalization software uses the concepts of writing systems and scripts as they are understood in the area of linguistics. This position differs from the one held in System 7, in which the concept of a script system and what composed one was particular to System 7. If you have relied on the understanding of these concepts imparted by descriptions of System 7 and its predecessor versions, you'll need to adjust your perspective somewhat to make the transition to international text support in Mac OS 8.

Writing Systems and Scripts As Understood in Mac OS 8

Mac OS 8 aligns with the standard view of writing systems and scripts expressed in linguistic literature and explained in this discussion.

Writing Systems

Written representation of a spoken language relies on a writing system. A writing system, then, is an artificial construct used to record language in written form. It can be viewed as having three main components—language, scripts, and orthography—with well defined relations to one another.

Scripts

A script comprises a set of symbols that represent the components of a language. A writing system uses one or more scripts for the symbols required to represent linguistic elements, which include sound, meaning, syntax and so forth. A script can be coupled with one language, or it can represent and be used by many languages. Moreover, a language can have more than one script associated with it. For example, the Japanese language uses the Japanese script, while the French, Italian, and Spanish languages all use parts of the Latin script.

A script exists apart from both the languages it represents and the writing systems for which it is used. (A small number of scripts, less than 100, are used by writing systems despite the large number of existing modern and archaic languages.) Scripts have largely developed in accord with geographical and cultural requirements; they show historical, linguistic, and geopolitical derivations and influences.

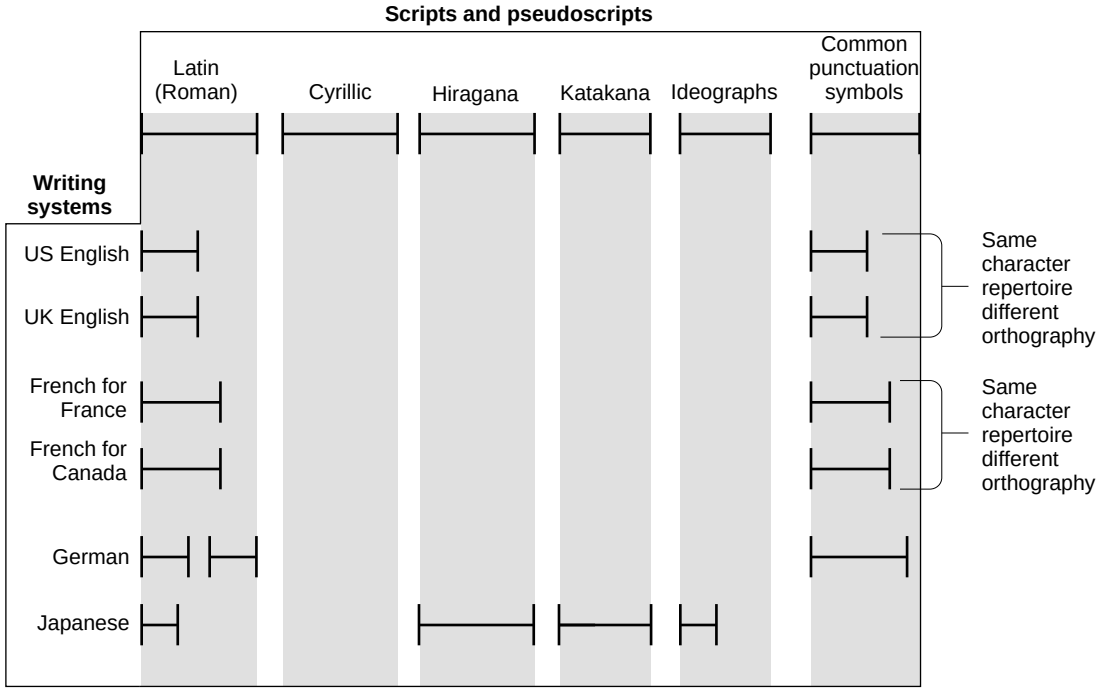
Some scripts are more inclined to represent sound, while others represent meaning, but usually scripts include both representations. Alphabetic scripts are thought to represent sound exclusively and hieroglyphic or ideographic scripts are thought to represent ideas, but this is a misconception because both systems include symbols for sound and meaning.

A special category of scripts, called *pseudoscripts,* exists for use with other scripts. These pseudoscripts include symbols, numbers, and punctuation.

Writing systems can use different scripts at the same time. A writing system uses at least one script and typically one or more pseudoscripts. In this sense, then, it is best to refer to the characters a writing system includes as a *repertoire* of characters, rather than a character set, because these characters can belong to different scripts. Figure 1-1 shows six writing systems and the scripts and pseudoscripts that they incorporate. Notice that although the U.S English and

U.K. English writing systems both use the same ranges of the Latin script and the common punctuation and symbols pseudoscript, they comprise distinct writing systems because their orthographies differ.

Figure 1-1 Writing systems, scripts, and orthographies



Orthographies

The writing system for a language entails an orthography which defines the relationship between the written language and one or more scripts.

Among the rules an orthography specifies are rules of directionality, level of discreteness, and units of representation. For example, for mixed-directional text, the direction of a paragraph is important. For writing systems based in European languages, a paragraph is considered a unit of representation, as is a word. Word division and paragraph identification are easily determined for

these languages, but this is not the case for writing systems based in Japanese or Indic languages.

Orthographies can differ between two geographical regions using the same language and collection of scripts. An example of this is the American English orthography, which differs from the British English orthography.

Text-handling system software that deals with the orthography of writing systems includes support for directionality, word breaking, hyphenating, and spell checking services, among others.

Writing Systems and Script Systems As Understood in System 7

For System 7, a writing system is described as a set of characters and the basic rules for their use in creating a visual depiction of language, rules for displaying, ordering, and formatting characters. Writing systems have specific requirements for text display, text editing, character set, and fonts. A script system is defined as a collection of resources, mostly tables of data, that defines the behavior of a particular writing system.

For System 7 a writing system breaks down into scripts, which imply locales and orthographies associated with locales. While the term text encoding is not used in System 7, a script in System 7 implies a text encoding, but it also implies additional data stored in international resources pertaining to locales. A script system can support various languages and regions. For example, the System 7 MacRoman script system supports the English, French, German, Italian, and Spanish languages. Within the French language are regional variations, for example, Belgian and French Canadian.

In System 7, often information stored in international resources is common to a group of locales (languages and regions) that use the script system. However, because of the way relationships are drawn between the scripts and the locales that particularize the data for the various writing systems or languages in System 7, international resources containing this data that applies to multiple locales is often replicated for each language or region. In System 7, a script connotes information about locales. This is not so in Mac OS 8.

Deconstructing the System 7 Script Manager and Looking at Mac OS 8 Solutions

The System 7 Script Manager provides exceptionally good international support compared with other available contemporary technologies. However, it is constrained by a number of limitations inherent in its design. For example,

text encoding is separated from the text to which it belongs. Also, extensibility in handling international resources is inelegant because the original design was envisioned to meet the requirement of a fixed number of resources.

In addressing and transcending these and other problems inherent in the Script Manager, designers of Mac OS 8 recognized the opportunity to press international support forward, set new standards, and realize in the design of new international text-handling and storage components goals of flexibility, extensibility, and ease of use.

This section explains some of the ways in which they did this. First, it describes the way the Script Manager addresses the complexities entailed in providing international text support, and then it describes how Mac OS 8 approaches these problems.

Text Handling and Storage in the System 7 Script Manager

This section identifies in System 7 how you store the encoding for text, how the system stores international data needed for text handling functions, and how you access that data. You can contrast this with the way these processes are handled for Mac OS 8 by looking at "Text Handling and Storage in Mac OS 8" (page 1-22).

- separation of encoding from text

System 7 multiscript support allows an application to handle text expressed in multiple languages, but it requires the application to store and manage script and language information used to represent the text apart from the text itself. In addition to adding complexity to code, this separation often results in the display of unreadable text when that text is moved from an application that provides multilingual support to one that does not, a condition sometimes referred to as *moji-bake*, a phrase that means character garbage in Japanese.

- overloaded script code

The System 7 Script Manager categorizes international writing systems with a data type called a script code. Depending on how an application uses them, script codes can signify multiple attributes, including language and region information, text encoding, and localization. Because script codes are multipurpose and overloaded, they are easily misused.

- storage of international data

For System 7, international data is packaged in a file type called a Script bundle. The system unpacks the data contained in this file and moves it into

the System file. The Script Manager was designed with the idea that a script would require a limited set of international resources— resources such as the numeric format resource, the long date and time resource, and the keyboard layout resource. In time, however, this group of resources proved to be insufficient and new resources were added, but they were handled in a variety of ways, including overloading the Script bundle and enhancing WorldScript to locate and load them.

- access to international data

For System 7, you use two routines—`GetIntResource` and `GetIntResourceTable`— for accessing international data stored in resources. These routines support a limited number of data types; adding new data types requires that these accessor routines themselves be modified. Resources your application can access using these routines follow a specific naming convention; if a resource does not follow this convention, to access its data, your code must read the resource directly from the system file.

Text Handling and Storage in Mac OS 8

This section identifies in Mac OS 8 how you store the text encoding specification for text along with the text, how the text encoding exists separate from the language and region information, how the system stores international data needed for text handling functions, and how you access that data. You can contrast this with how these processes are handled for System 7 by looking at "Text Handling and Storage in the System 7 Script Manager" (page 1-21).

Mac OS 8 separates information previously coalesced in System 7; it provides a distinct text encoding specification data type for identifying the text encoding or text encoding scheme and other information used in representing text; and it provides a distinct locale identifier data type specifying the language and region information used to characterize text or collections of data for specific writing systems or languages. These locale-specific data are stored in the locale database separately from the files containing text encodings.

- coupling of text encoding specification and language and region information with text

Mac OS addresses problems that result from storing information used to represent text apart from the text itself by encapsulating in a text object the text string and all pertinent information about text representation. Text objects simplify the work you need to do to associate text encoding information with text. Along with a text string, a text object stores the string's text encoding specification, its language and region information, and any annotations for it. A

text object can contain multilingual text with text runs carrying this information for varying text segments. The text, its encoding, its language and region information, and its annotations remain together as the text is moved from one application to another, diminishing the possibility that your text will be displayed in an unreadable manner. All of the information, including annotations, stays together as text is cut, copied, and pasted within your application.

IMPORTANT

Annotations are similar to System 7 resources in that the semantics of an annotation are available to applications and system components that understand the annotation's particular tag type. Although applications other than the one that created the annotation and Mac OS 8 system components might be able to interpret the semantics of an annotation, you should not assume that they do or that they will preserve an annotation's semantic integrity. ▲

If you use text objects, the system performs any necessary encoding conversions for you. For example, if your application performs collation processes and you use text objects, the system will convert text expressed in any text encoding to Unicode—if the strings are in different encodings or if collation tables don't exist for the original encoding—so that all strings are compared in the same encoding. See "Text Objects for Text Storage and Interchange" (page 1-25) for more information on text objects. If you do not use text objects, your application can convert text across encodings using one of the encoding converters. See "Text Encoding and Conversion" (page 1-67) for more information on the Mac OS 8 encoding converters.

- separation of text encoding from language and region

Instead of bundling together data for a geographical region with the text encoding used to represent the text, the design of Mac OS 8 separates the text encoding or text encoding scheme from data used for the orthography of the language and region for which the text is to be localized, and from the font used to image the text.

In addition to the standard set of Mac OS 8 text encodings and text encoding schemes shipped with the software, third-party developers can provide their own text encodings and schemes, extending the range of possible ones your application can support. Moreover, Mac OS 8 supports the Latin-1(ISO 8859-1) text encoding, which is currently the most common one used for the Internet,

so you can support text files in this encoding that your user obtains from the Internet.

Resources containing data used to localize text for a certain geographical region are stored in the locale database, whereas information for text encodings and schemes is stored separately in files containing the coded character set and all information needed for its mapping and conversion to Unicode.

This separation, which makes writing systems independent of text encodings and text encoding schemes give you more control over how your application can present choices in its user interface.

- storage of international data

The Mac OS 8 locale database provides a way in which you can store any type of data used for text handling and text-behavior-setting, and easily modify that data. The locale database clusters together such data for a specific geographical area according to its locale and region information. These clusters are called locales and each one is composed of locale objects. A Mac OS 8 locale object is roughly equivalent to a System 7 international resource.

The locale database is highly extensible; the kind of international preferences data and other data known to be required today does not limit or define what you can store in the locale database and access as future requirements surface.

You can permanently add data to the locale database for use by all applications and system components, or you can extend the data available for your application's use within its current process by temporarily adding data to the database. You can also temporarily override the default behavior of various text-handling operations for the language of a specific geographical region from within your application's current process to customize it for your use.

The Locale Object Manager creates the locale database and adds data to it at system startup, but system components and your application software can determine which data to use—for example, for localization—at runtime after determining the user's preferred language and the appropriate character set for that language.

The locale database offers an extensible means of storing international preferences data. To provide backward compatibility with System 7, the Mac OS 8 release of the locale database provides a way to incorporate System 7 international resources, making those resources available to your application; they show up in your resource chain, just as they always did with System 7, even though they are stored in the locale database.

- access to international data

You use the Locale Object Manager to access data stored in the locale database. Each locale object belonging to a locale is a separate entity that you can access independently. All data that exists in the locale database is cataloged along the same lines and accessed using the same method. Moreover, data stored in the locale database is cataloged along multiple lines based on information describing that data. When developers create locale objects, they provide various kinds of information describing the data. The Locale Object Manager uses this information to classify and catalog the data; it includes this defining information with the data when it adds the data to the database.

This way of cataloging locale object data allows you to access data of a certain type for various languages and regions by specifying any of its characteristics. For example, you might want to find all input methods for languages that use 2-byte character encodings, or you might want to find all data-and-time formatting data for a specific language. The Locale Object Manager locates and returns to you any data resident in the database that meets a set of specifications you provide.

While the locale database offers extensibility, the Locale Object Manager offers ease of use and flexibility in accessing that data. For more information on how to access data in the locale database, see "Storage and Retrieval of International Data and Preferences" (page 1-48).

Text Objects for Text Storage and Interchange

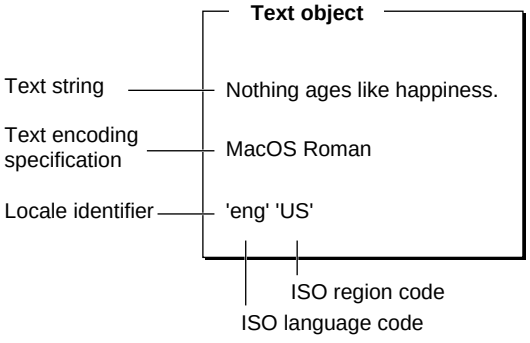
A **text object** is the fundamental unit of text interchange in Mac OS 8. You use text objects to pass text to or receive text from system components. For example, to specify a name to the file system, you use a File Manager function, passing it a text object containing the name. You also use text objects to specify text that is displayed as part of your application's user interface. All user interface elements in your application that contain localizable text should specify the text using text objects and store the text in text object resources.

A text object consists of a **text encoding specification**, which identifies the text encoding in which the text is expressed, a locale identifier, which identifies the locale in which the text was originally created, and the actual text itself. A locale identifier encapsulates an International Standards Organization (ISO) language code, which specifies the language in which the text is to be

represented, and an ISO region code, which specifies the geographical region for languages that vary by region. A locale identifier can also contain a customization code, but these codes are not retained by text objects because the custom settings become invalid or obsolete as the locale database is rebuilt or changes.

Figure 1-2 shows a conceptual rendering of a simple text object's contents.

Figure 1-2 A simple text object



Note
A text object can also contain annotations, not shown in Figure 1-2. Annotations are discussed later in this chapter.
◆

Because they enclose the encoding specification, and language and region information along with the text, text objects make it possible for software that did not create the text to process it correctly in an environment in which multiple text encodings and languages are used. For example, the user interface elements of an application localized for Hebrew will be depicted in the Hebrew language on a U.S. MacOS Roman system if the MacOS Hebrew character set and corresponding glyphs used to represent the text are available on that system.

Text Object Contents

This section describes the primary contents of a text object from a conceptual perspective; these are the parts of a text object your application provides or manipulates using the Text Object Manager. A text object contains other information used internally, which is not described here.

A text object encapsulates

- the text string.
- the text encoding specification giving the text encoding used to express the text. See "Text Encoding and Conversion" (page 1-67) for information on text encodings and specifications.
- the locale identifier consisting of the ISO language and ISO geographical region codes identifying the language and region for which the text is localized. For background information on locale identifiers, see "Storage and Retrieval of International Data and Preferences" (page 1-48).
- one or more optional annotations used to mark the whole text string or segments of it with any additional information you want used in conjunction with that text. See "Text Annotations" (page 1-39) for information on annotations.

A Simple Text Object

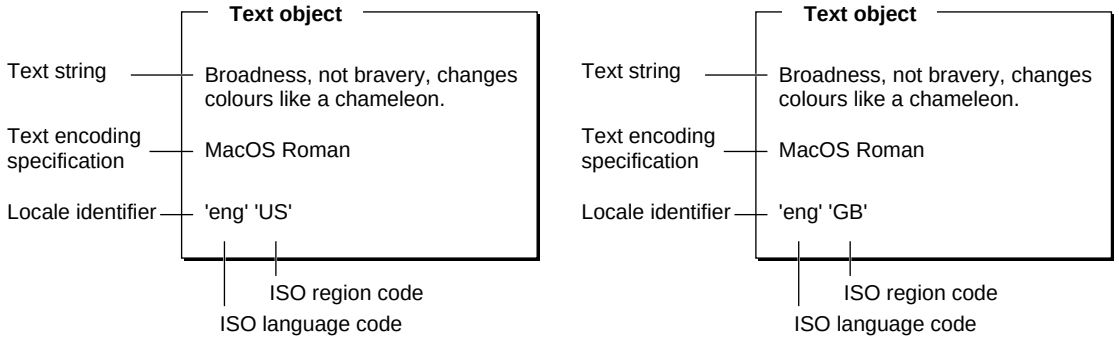
A text object can contain one or more text runs. A **text run** is a text string segment that is characterized by a single text encoding specification and locale identifier stipulating how the text is to be represented. When the text object's entire text string has a single text encoding specification and locale identifier associated with it, the object contains a single text run.

A locale identifier includes an ISO language code and, if applicable, an ISO region code. A language code is a three-character, lowercase identifier used to indicate a particular written version of a language for Mac OS 8. A region code is a two-character, uppercase identifier used to indicate a version of the written language of a particular region or territory.

Mac OS 8 recognizes the language codes defined by ISO in the ISO CD 639/2 draft proposal titled "Code For the Representation of Names of Languages, alpha-3 code" dated December 16, 1991. Constants defined for these codes are included as comments in the `TextCommon.h` file.

Figure 1-3 shows two separate simple text objects, each containing the same text string. The first text object has a MacOS Roman text encoding and a locale identifier for the English language of the United States. The ISO language code 'eng' specifies the English language. The ISO region code 'US' specifies the geographical region of the United States. The second text object has a MacOS Roman text encoding and a locale identifier for the English language of Great Britain. Variations associated with the English language as written and spoken in the United States apply to the first text object while variations associated with the English language as written and spoken in Great Britain apply to the second text object. For example, a spelling checker created for the United States region would indicate that the word *colour* in the text string is misspelled and suggest the regional spelling *color*.

Figure 1-3 Two single text-run text objects for different regions



A Text Object Containing Multiple Text Runs

The text string of a single text object can be composed of multiple text runs. This is the case if various segments of the text have associated with them different text encoding specifications and locale identifiers. Figure 1-4 shows a text object whose text string contains these three text runs:

- The text segment “The old believe everything” is represented in the MacOS Hebrew text encoding and the Hebrew language of Israel. The ISO language code 'heb' specifies the Hebrew language. The ISO region code 'IL' specifies the geographical region of Israel. An annotation specifying the color blue is

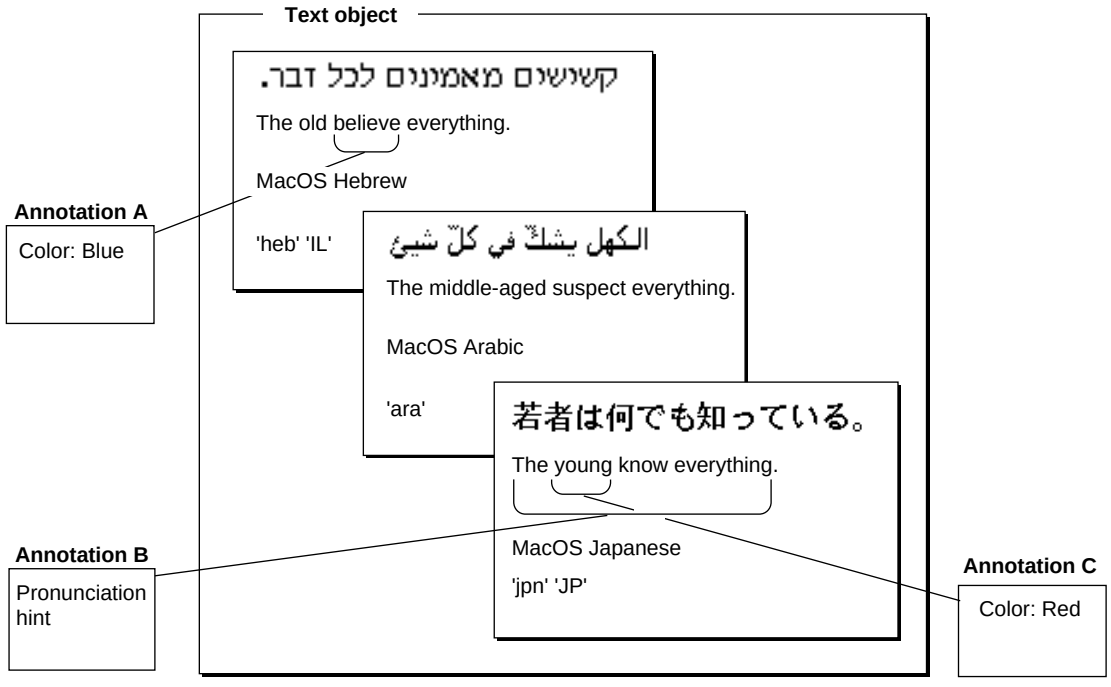
associated with the word “believe.” The Text Object Manager imaging functions would not interpret the annotation or image the text in blue. The application might use the color information stored in the annotation with another imaging system.

Note
Recall that although the Mac OS 8 system software preserves annotations, it does not interpret their content. ◆

- The text segment “The middle-aged suspect everything” is represented in the MacOS Arabic text encoding. The ISO language code 'ara' specifies the Arabic language. There is no annotation associated with the text of this text run. (When you create and use locale identifiers, you can use the locale region wildcard, `kLocaleRegionWildCard`, along with the ISO language code to specify that you want to use the standard form of a language, and not a particular regional form of the language.)
- The text segment “The young know everything” is represented in the MacOS Japanese text encoding and the Japanese language of Japan. The ISO language code 'jpn' specifies the Japanese language. The ISO region code 'JP' specifies the geographical region of Japan. An annotation providing a pronunciation hint is attached to the entire text segment, while a color annotation is attached to the word “young” only.

Figure 1-4 shows this text object.

Figure 1-4 A text object with multiple text runs



How Text Objects Are Used

Because they facilitate internationalization of an application, you should use text objects for all parts of your application's user interface, for example, for static text in menus, panels, and dialog boxes. However, text objects are not meant to be used as a document model, that is, for text-intensive applications such as word processors.

The Text Object Manager provides an application programming interface (API) that you can use to create, modify, and dispose of text objects, obtain their contents, and obtain information about them. You can copy and replace the contents of text objects with a text string or contents from another text object, append text to an existing text object, and concatenate two text objects to create

a third one combining them. You can obtain the text contained in a text object as a text string. You can attach annotations to the text segments of a text object's text string and replace and delete existing annotations.

You can determine whether a text object is empty, get its size, determine the number of text runs it contains, and obtain the encoding of a portion of text within a text object. You can convert text objects to and from Unicode, Pascal, or C strings.

The Text Object Manager also includes a set of imaging functions that you can use to draw the text of a text object.

Text Objects and Text Strings: A Comparison

A text object differs from a simple text string in three primary ways:

- A text object encapsulates and carries the text encoding specification and the language and region information used to represent the text along with the text string.
- A text object does not allow direct manipulation of the text. Instead, you use the Text Object Manager functions to extract text from or put it into a text object. A text object stores the text string and its attendant information in a private data structure, and, is therefore, opaque to your code. By hiding the details of the text encoding specification from your application, text objects provide for an easy transition to a system based in Unicode.
- A text object can include annotations that are associated with the whole text string or portions of it. Annotations can contain whatever additional data you want to associate with the text string. You can use annotations for any purpose suited to your application. See "Text Annotations" (page 1-39) for information on annotations.

Text objects provide functionality equivalent to that of C strings and Pascal strings. C string and Pascal string representations used as the principal means of expressing text in System 7 and earlier versions of the system software do not lend themselves to Unicode. C strings are not feasible because they are null terminated and null bytes occur in many Unicode character encodings. Pascal strings are simply too short to hold enough Unicode characters to be useful.

Text Object Types

There are two kinds of text objects distinguished mainly by how storage for their contents is managed: **ephemeral text objects** and **persistent text objects**.

The Text Object Manager dynamically allocates and initializes the memory for an ephemeral text object when your application calls the Text Object Manager function that creates one. The Text Object Manager manages the memory for an ephemeral text object, expanding and contracting the text object as necessary to accommodate the text and modifications to it.

You provide a block of contiguous memory for a persistent text object; the Text Object Manager will use only that memory for the persistent text object. A persistent text object is self contained; you can move a persistent text object around and preserve it until you no longer need it. You can either stack-allocate a persistent text object or create one in a fixed-size data structure. You should always use a persistent text object if the text object will be passed from one address space to another.

Your application allocates the memory for a persistent text object in text object units and passes that memory to the Text Object Manager function, which you call to initialize the persistent text object. For this purpose, the Text Object Manager defines the `TextObjectUnit` data type, which consists of 4 bytes, naturally aligning on a longword boundary.

You use a pointer of type `TextObject` with the Text Object Manager functions to point to the beginning of a text object, whether the text object is an ephemeral or persistent one.

Indices for Text in a Text Object

The Text Object Manager allows you to manipulate the text in an existing text object. You can extract and copy text from a text object; you can insert text in and append it to a text object; you can replace text in a text object with other text. Functions for these and other purposes require that you identify the character or text segment of the object's text string that you want to affect. You use text object indices for this purpose.

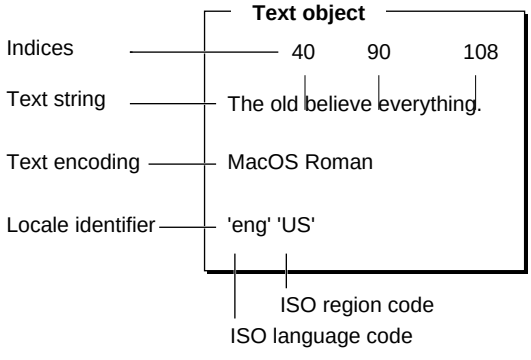
The position of a character within a text object is indicated by an index. A **text object index** is a number that indicates a position between characters, before the first character, or after the last character of the text in a text object. A text object index does not refer to a specific character, rather it indicates the position

before or after that character. The Text Object Manager uses indices to identify the position of characters composing an object's text string.

Usually, you use a set of text object indices to specify a portion of the text within a text object; to do so, you identify the index position before the character that begins the text range and the index position after the character that ends the text range.

The numerical values of text object indices are associated with the specific internal representation of a text object's text, so they are not valid across Text Object Manager functions that modify the text object's content. Numbers of indices increase monotonically within a text object, but they are not necessarily sequential. If the text of a text object has been modified since you last obtained indices for a segment of its text, and you want to refer to that text segment again—for example, to copy it again—you should use a Text Object Manager function to obtain the new indices delineating that text segment. The characters of the text string shown in Figure 1-5 are marked with indices identifying their positions. Notice that the numbers representing the indices increase, but they are not sequential (nor do they correspond to byte offsets).

Figure 1-5 Text object indices



The Text Object Manager provides constants for referring to the beginning of a text object and its end. It is always safer to use these constants when you want to specify the beginning or end of the text of a text object instead of attempting to calculate the index values. These constant will always refer to the beginning and end, while your calculated index values may not.

Imaging With Text Objects

The Text Object Manager includes a set of Color QuickDraw–based functions that you can use for imaging text objects. Unlike the System 7 QuickDraw Text functions, the text object imaging functions allow you to specify the graphics port to be used. These functions take an explicit color graphics port parameter instead of using the current graphics port. You can measure and draw text using these functions. The functions return any measurements they compute as fixed integers instead of simple integers, as was the case for the System 7 functions.

To draw the text of a text object, you call the `DrawTextObject` function. `DrawTextObject` draws the text at the current pen position using the glyphs for the language and text encoding specified by the text object. The pen is left at the end of the imaged text. When you call this function, you can provide it with a font selection hint to be used if the Text Object Manager must perform font substitution.

The text object imaging functions take a global text direction parameter for handling multidirectional text, which you can set to left-to-right or right-to-left or base it on the current system default. To give you more flexibility in the use of these measuring and drawing functions, the Text Object Manager defines imaging options. You can set the bit flags in a function's option bits parameter to specify aspects such as alignment, justification, and handling of text too wide for the available space. Standard behavior, which you can accept or override, is defined for each function.

Font Selection Hint for Font Substitution

A text object can contain multiple text runs, each of which is composed of text to be represented in a different language and text encoding from its adjacent text segments. When you draw or measure text, the specified text segment may span multiple languages and encoding systems. When it does, the Text Object Manager must determine the correct font to use for each text run the text segment contains.

The Text Object Manager offers an interim solution to the problem entailed in determining the most appropriate font to use in imaging text objects containing multi-encoded, multilanguage text. For this purpose, the Text Object Manager defines tokens representing special fonts. You can select one of these tokens and pass it to an imaging function as part of the function's option bits parameter. The token provides a font-substitution hint to the Text Object Manager when the current font is not the most appropriate one to use to image

the text of a text run. This hint will direct a text object imaging function to try using the particular special font defined for the required text encoding of that text run. If the Text Object Manager cannot find an appropriate font, it will use the current one.

IMPORTANT

Again, this is an interim solution for this release of Mac OS 8. When a common, system wide solution to the font-selection problem is available, a developer release will address it. ▲

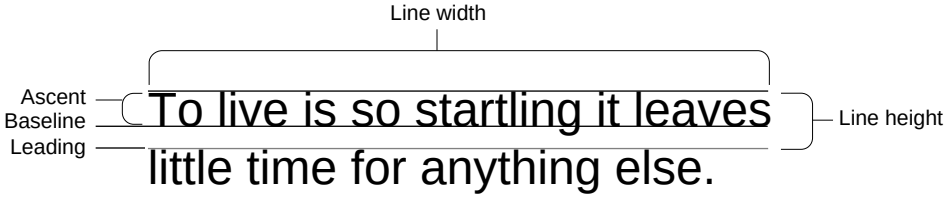
Text Measurement

One of the imaging functions provided by the Text Object Manager returns three metrics associated with imaging the text of a text object. The function calculates and returns these three measurements in pixels as fixed-point values:

- the width of the text object as imaged by the `DrawTextObject` function. You can use the width for performing tasks such as highlighting.
- the total line height taking into account any font substitution; line height is the measurement or the vertical distance from the top of the text (or the ascent line of the text) down to the bottom of the leading beneath the text (or the ascent line of the next text line).
- the ascent, that is, the distance from the baseline to the top of the text.

Figure 1-6 maps these metrics to an imaged text string.

Figure 1-6 Text imaging metrics



Text Alignment and Justification

The Text Object Manager allows you to specify text alignment and text justification separately when drawing text in a rectangular region defined by a box. **Alignment** is the horizontal placement of lines of text with respect to the left and right edges of the text area. **Justification** is the spreading or compressing of printed text to fit into a given line length so that it is flush on both left and right edges of the text area.

When you use the Text Object Manager imaging functions, you can specify justification separately from alignment to allow for handling the last line of text in a paragraph. If you turn on justification, the Text Object Manager functions will justify all of the text of a paragraph except for the last line. The last line will be aligned according to the method of alignment you specify, but not justified.

You can specify that text be left aligned, centered, or right aligned. Figure 1-7 shows three examples using the same text: for the first one, the text is justified and left aligned, notice that the last line in each paragraph is left aligned, not justified; for the second one, the text is justified and centered, notice that the last line in each paragraph is centered, not justified; for the third one, the text is justified and right aligned; notice that the last line is right aligned but not justified.

Figure 1-7 Text alignment and justification

Justified with last line aligned left.

Difficult and easy complement each other.

Long and short contrast each other.

High and low rest upon each other.

Justified with last line aligned center.

Difficult and easy complement each other.

Long and short contrast each other.

High and low rest upon each other.

Justified with last line aligned right.

Difficult and easy complement each other.

Long and short contrast each other.

High and low rest upon each other.

Controlling Text Flow When the Text Is Too Wide for the Line

When you draw text on a line using the `FlowTextObjectOntoLine` function, you can exert finer control over how the text is to be handled if it is too wide to fit on a line, rather than accepting the default treatment. By default, the function first tries to condense the text, then truncates it, and finally clips the text image.

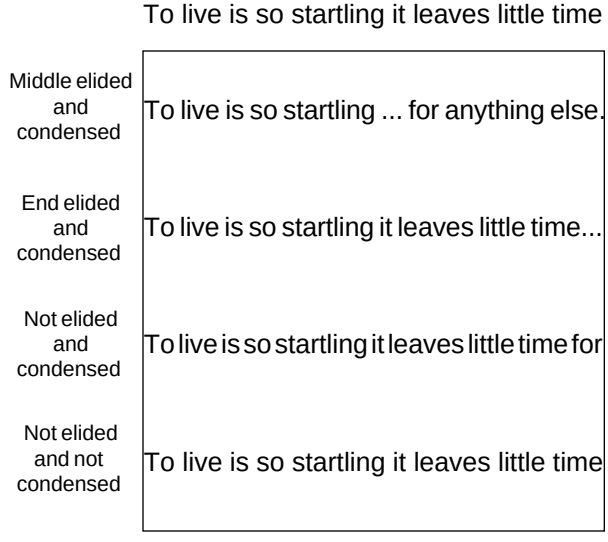
To refine how the text is handled, you can specify whether it should be condensed, elided, or both. If elided, you can specify whether the text should be elided in the middle, at the beginning, or at the end.

Note
Unlike the System 7 Script Manager's `TruncateText` function, `FlowTextObjectOntoLine` does not modify the original text object. All of the original text remains the same. ◆

If you specify the condensed option, the Text Object Manager will use a condensed font to fit the text on the line. If you specify that the text is to be elided, the Text Object Manager will elide the text by omitting a portion of it where you specify—at the beginning, middle, or end—and replacing the text with an omission symbol, such as the ellipsis, specific to the language in which the text is represented.

Figure 1-8 shows an example in which the first two lines are both elided and condensed. The first line is elided in the middle, and the second line at the end. The third line is condensed but not elided. The last line is neither elided nor condensed.

Figure 1-8 Condensing and eliding text



Text Annotations

The Text Object Manager provides a way for you to associate related data with the text of a text object.

You associate related data with text in the form of an annotation. You can use annotations for any purpose suited to your application. For example, you can use annotations to tie pronunciation hints for text-to-speech conversion to the text string. In handwriting recognition systems, you can store the "ink" version—what the user actually writes on the screen—with the textual version using annotations.

You might want to use annotations, for example, if you want your application to be able to sort files in languages that do not use alphabets, such as Japanese. For example, without the use of annotations, there is no convention that you can use to sort a list of names if the names were created with the Japanese coded character set. However, if you attach the phonetic pronunciation of each name to the text for a name, you can sort according to pronunciation. In handwriting recognition systems, you can store the "ink" version -- what the user actually writes on the screen -- with the textual version using annotations.

You can annotate a text object's entire text string or any portion of it. You can attach one or more annotations to a text string. You can associate multiple annotations containing different kinds of data with all of the text of a text string or annotations containing the same or different data with different segments of the text string; you can overlap annotations across contiguous segments of text.

There are two ways to add annotations to a text object:

- You can explicitly apply an annotation to the text of a text object.
- You can replace text in one text object with text from another, and the annotations attached to the replacement text are carried along with it.

By design, text objects do not carry style information; rather, the imaging system that your application uses is responsible for providing the style information for the text of a text object. However, it is possible for you to annotate text with style information. If you do, the semantics of the style information are private to your application. That is, your application is responsible for interpreting the style information; the Text Object Manager does not intervene in any way.

Annotation Types and Storage

You use the Text Object Manager functions to attach an annotation to an object's text segment. A text object annotation is a single block of data that is self-contained and does not refer to other areas of memory. For example, you should not include embedded pointers in any annotation that you provide. If the text object containing the annotation were moved to a new address space, any pointers that it contained would be invalid. When you provide and attach an annotation to a text object, the Text Object Manager copies the data into the text object.

Text object annotations are distinguished by type. An annotation's type is represented by a 4-byte tag. You define annotation types to identify the kind of annotation data you supply. Annotation tags follow the rules that apply to 4-byte identifiers. An annotation type tag can be any sequence of uppercase ASCII letters. Apple reserves for its own use lowercase ASCII letters, all spaces, or all international characters (characters greater than $7F). The Text Object Manager defines a wildcard annotation type that you can use to affect all annotations of a text object—for example, to delete them all.

IMPORTANT

You must register any annotation types and their tags that you define with the Apple Developer Support Center (`applelink:DEVSUPPORT`). (This procedure is similar to the one you follow in registering creator and file types.) ▲

Annotation Syntax and Semantics

An annotation's data is meaningful to your application only; the Text Object Manager does not interpret it. The Text Object Manager ensures that the syntax of annotations remains valid across changes to the text object. Ensuring the **syntactic validity** of an annotation means that the Text Object Manager guarantees that an annotation's size and data will not change as it adjusts the range of text to which that annotation applies. It also means that the Text Object Manager adjusts the annotations so that they continue to apply to the correct portions of the text string and it ensures that annotations of the same type do not overlap.

After the text has been modified, annotations apply to the same text as they did before, though the textual regions may have changed; for example, some of the text might have been deleted. The Text Object Manager also ensures that annotations do not apply to any new text added to the text object. For more

information on this process, see "How Annotations Are Adjusted When Text Is Modified" (page 1-42).

The Text Object Manager does not ensure the **semantic validity** of annotations within the text object after modifications to it. The responsibility for this lies with your application. Semantic validity refers to the internal meaning of an annotation in relation to its text.

In Mac OS environment, text objects can pass across address spaces and between different computers. The Text Object Manager is present and acts on text objects in these circumstances, ensuring their syntactic validity but not their semantic validity.

Annotations are similar to System 7 resources in that the semantics of an annotation are available to applications and system components that understand the annotation's particular tag type. Although applications other than the creator of the annotation and Mac OS 8 system components might be able to interpret the semantics of an annotation, you should not assume that they do or that they will preserve an annotation's semantic integrity. Your application should always be prepared to validate any annotations it has created or is able to interpret if the text object containing the annotation is passed to another application or system component that might modify it.

If your application has multiple threads sharing access to text objects, it's your application's responsibility to protect access to the text objects.

Annotation Attributes

Whenever a text object has been modified, your application is responsible for ensuring the semantic validity of annotations within the text object. When it modifies a text object, the Text Object Manager sets attribute bits in the annotations of the text object. Each annotation contains two attribute bits. These bits serve as hints, indicating that you may need to validate an annotation's semantics. Here is how you can interpret the bits:

- You can think of the text-object-annotation-changed attribute bit as signifying a local change. That is, the Text Object Manager sets this bit in an annotation when it modifies the range of text to which the annotation applies.
- You can think of the text-object-text-changed attribute bit as signifying a global change, a change somewhere in the text of the text object. That is, the Text Object Manager sets this bit in every annotation of the text object when it modifies any of the text of a text object.

For example, suppose a text object contains a text string for the phrase "Changing the world one person at a time." In this scenario, suppose one annotation is attached to the text segment "Changing the world" and another annotation is attached to the text segment "time." Suppose you replace the characters for the word "changing" with the characters for the word "seeing."

Here is how the Text Object Manager would set the annotation bits in the annotations attached to these text segments after changing the text:

- It would set both bits in the annotation that now applies to "Seeing the world." It sets both bits because the text to which the annotation applies has been changed; this modification also qualifies as a change to any of the text in the text object.
- It sets only the text-object-text-changed attribute bit for the annotation that applies to "time" because text elsewhere in the text object has been changed but the local text to which the annotation applies has not.

You can use a Text Object Manager function to check an annotation's bits to determine whether you need to update the annotation's contents to ensure its semantic validity. The function that allows you to obtain information about the annotations of a text object returns data structures containing an attributes field for each annotation. This field includes two bit flags representing the annotation bits. The Text Object Manager provides constants that define masks you can use to test these bit flags. After you validate the annotation data, you can use another Text Object Manager function to clear one or both of the annotation attribute bits for annotations of a particular type. This allows you to reuse the bits; you can refer to them later to see if the text has been changed again.

How Annotations Are Adjusted When Text Is Modified

The Text Object Manager follows these rules in adjusting annotations after modifying the text of a text object:

- Only one annotation of a given type can exist for a text string segment within a text object. That is, no annotations of the same type can have overlapping ranges.
  - Adding an annotation deletes any annotations of the same type that fall entirely within a new annotation's range.
  - Any older annotations overlapping the text segment of a new annotation of the same type will be adjusted so that they no longer overlap the new annotation's range.

- All annotations are cloned when their host text is split. The cloned annotations are identical to the original except that the original and the clone now apply to separate text segments.

The Text Object Manager makes these four kinds of adjustments to annotations when the text of a text object is changed in some way, depending on how the text is modified:

- If the text range that an annotation spans is completely deleted, so is the annotation. For an example of this, see Figure 1-9 (page 1-45).
- If an annotation's range completely contains a deleted text region, the endpoint of the annotation's range is adjusted to reflect the deleted text.
- If an annotation's range intersects a deleted text region but neither contains it nor is contained by it, the annotation's end point in the deleted text region is adjusted to be outside it. That is, the annotation's range shrinks so that it longer intersects the deleted region. Instead, it now applies to the remaining portion of the text range to which it originally applied. For an example of this, see Figure 1-9 (page 1-45).
- If text is inserted into the region spanned by an annotation, that annotation is split so that the annotation is attached to the same text as before the insertion, which is now two separate segments. For an example of this, see Figure 1-10 (page 1-47). Here is how the Text Object Manager effects this:
  - It adjusts the original annotation's endpoint so that the annotation does not span the inserted text.
  - It copies the annotation and its data and attaches it to the remainder of the original text segment that comes after the inserted text.

Effects of Replacing, Inserting, and Deleting Text on the Text and Its Annotations

You can replace a portion or all of the text of a text object with new text. You can insert text into a text object and delete text from one. Whenever you take these actions, annotations associated with the text are affected.

Any annotations associated with text you *insert* into a text object are carried along with the text. Annotations associated with text you *delete* from a text object are adjusted to accommodate the remaining text to which they apply. To replace text, the Text Object Manager first deletes the text to be replaced and then inserts the new, replacement text.

For text deletion, insertion, and replacement, the Text Object Manager follows the rules governing the four kinds of adjustments made to annotations when text is modified. See "How Annotations Are Adjusted When Text Is Modified" (page 1-42) for information on these rules.

Text Deletion and Annotations

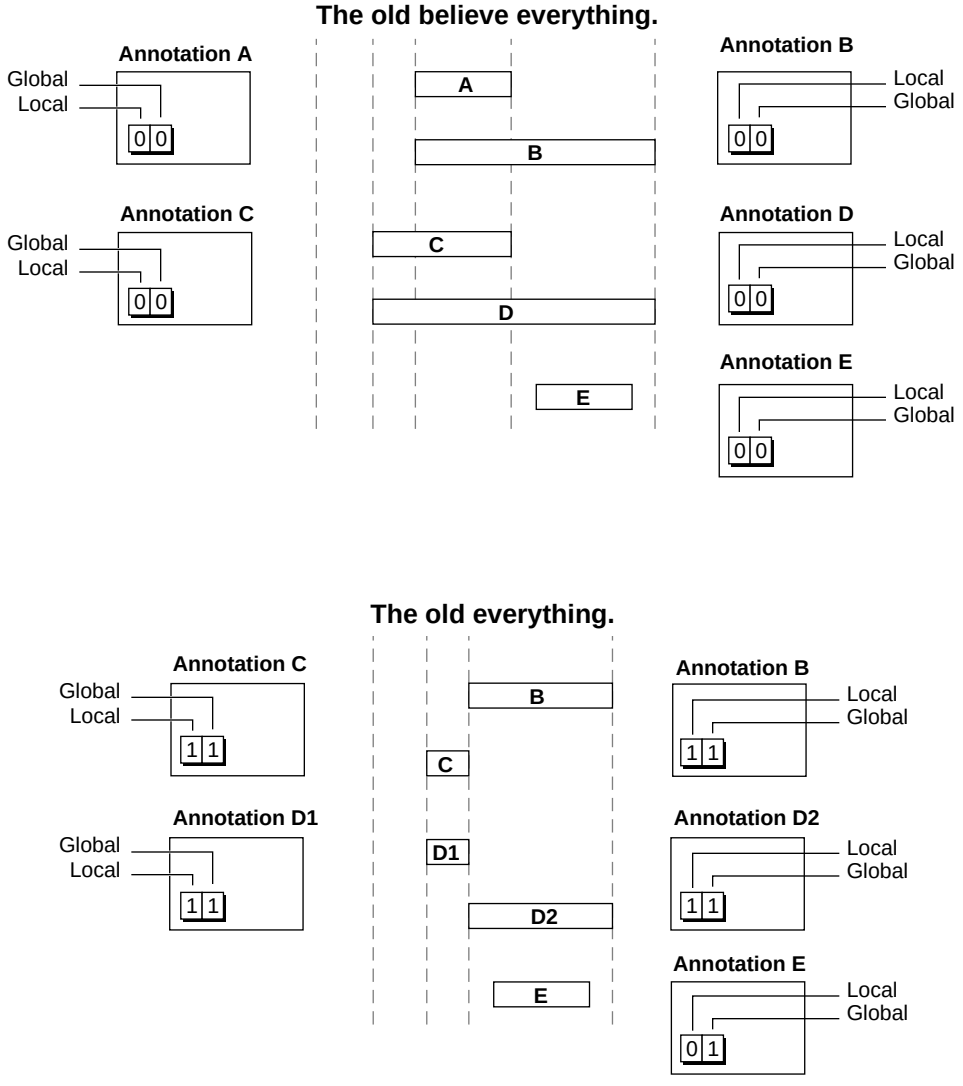
In the example shown in Figure 1-9, the word "believe" is deleted from the text object that encapsulates the text string "The old believe everything" and its associated annotations.

These four annotations are associated with the word "believe".

- The first one (A) applies to the word "believe" exclusively, so it is deleted entirely.
- The second one (B) applies to both the word "believe" and the word "everything", so it is adjusted to apply to the word "everything" only. Notice that both the local (text-object-annotation-changed attribute bit) and the global (text-object-text-changed attribute bit) bits are set for annotation B.
- The third one (C) is adjusted to apply to the word "old" only, and both of its bits are set.
- The fourth one (D), which spanned the text segment "old believe everything" is split into two annotations, each containing its own set of dirty bits, to now apply to both "old" and "everything" and both bits are set.

Notice that annotation E originally applied only to the word "everything", so this annotation was unaffected by the deletion. For that reason, only its global (text-object-text-changed attribute bit) bit is set.

Figure 1-9 Effect of text deletion on annotations

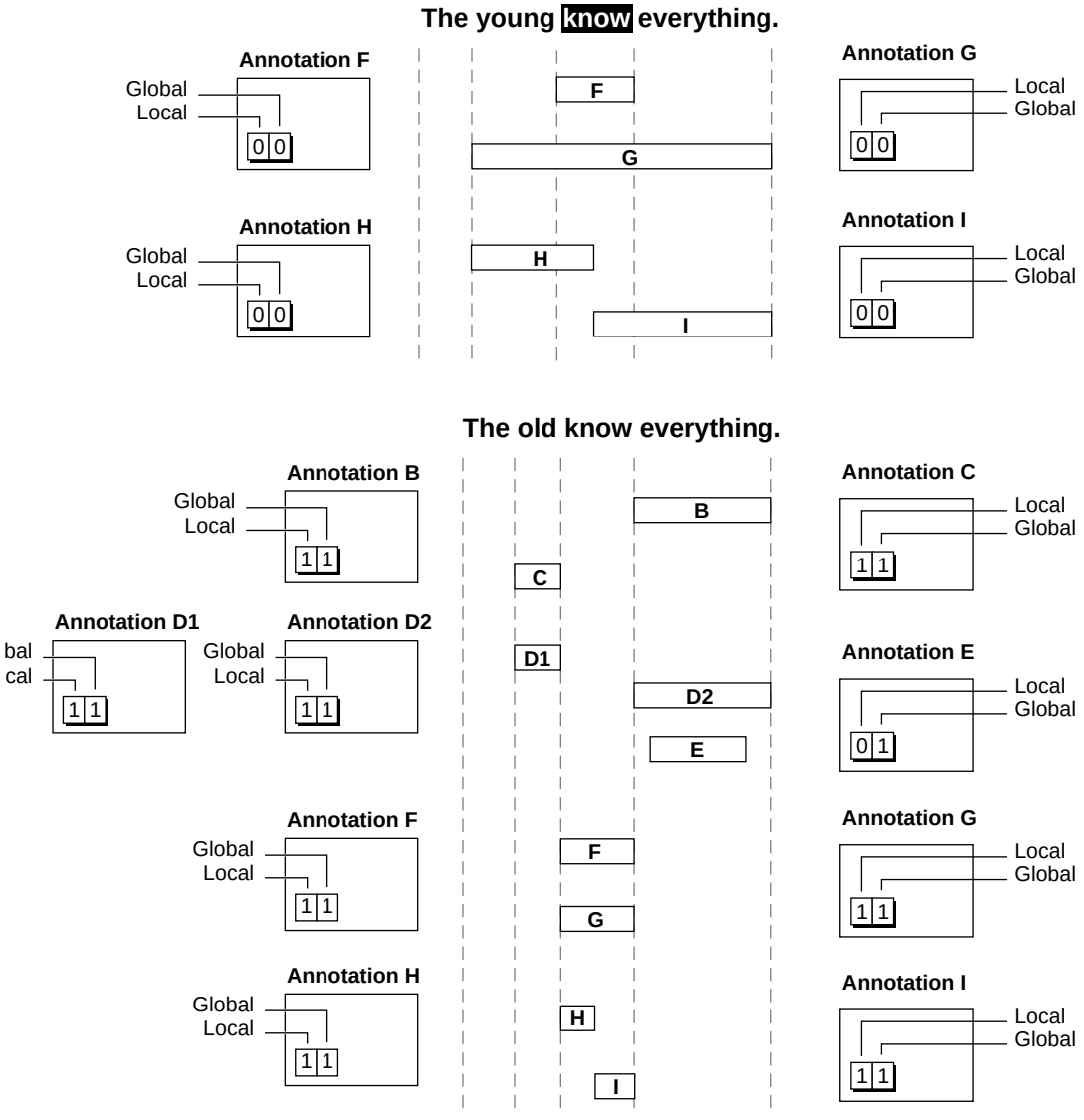


Text Insertion and Annotations

Using a text object containing the text string “The young know everything,” and the text object from the deletion example discussed in “Text Deletion and Annotations” (page 1-44), assume that the word “know” is copied and pasted into the text object whose word “believe” was deleted at the same place. The resulting text string in the text object now reads “The old know everything.” Figure 1-10 shows the text object containing the text string “The young know everything.” and its annotations, and the resulting text object containing the text string “The old know everything.” As shown in Figure 1-10, annotations are adjusted in the following way to produce the resulting text object:

- Annotation F applies entirely and only to the word “know”, and it is copied and carried along with the word so that it applies to it, and only it, in the text object resulting from the insertion.
- Annotation G is copied and shrunk to apply to only the word “know” when the word is inserted in the resulting text object.
- Annotation H is copied and adjusted to apply to only the letters “k” and “n” in the resulting text object.
- Annotation I is copied and adjusted to apply to only the letters “o” and “w” in the resulting text object.

Figure 1-10 Effect of text insertion on annotations



Text Replacement and Annotations

From your application's perspective, text replacement appears to be a single process. In fact, to perform text replacement, the Text Object Manager first performs a deletion and then an insertion. The processes discussed earlier in "Text Deletion and Annotations" (page 1-44) and "Text Insertion and Annotations" (page 1-46) perform text replacement.

Storage and Retrieval of International Data and Preferences

International preferences and other data— data such as date-and-time strings, number formats, hyphenation dictionaries, and collation schemes—define in part the orthography for a particular writing system. This data allows for language or regional variations within a writing system.

Mac OS 8 components that address international text requirements use this data to determine how to handle text for the world's various scripts. Whenever your application uses an application programming interface (API) to one of these components, your application indirectly uses this information. Your application might also have occasion to use this data directly. For example, you might want to obtain date-formatting information for a specific language to display a date to your user in a particular format belonging to that language. You might even want to customize the behavior of a certain operation, such as collation, for a particular language.

To address the storage and access requirements for data used for international text handling, Mac OS 8 includes a database called the locale database; to give you access to data stored there, it includes a component and its API, called the **Locale Object Manager**.

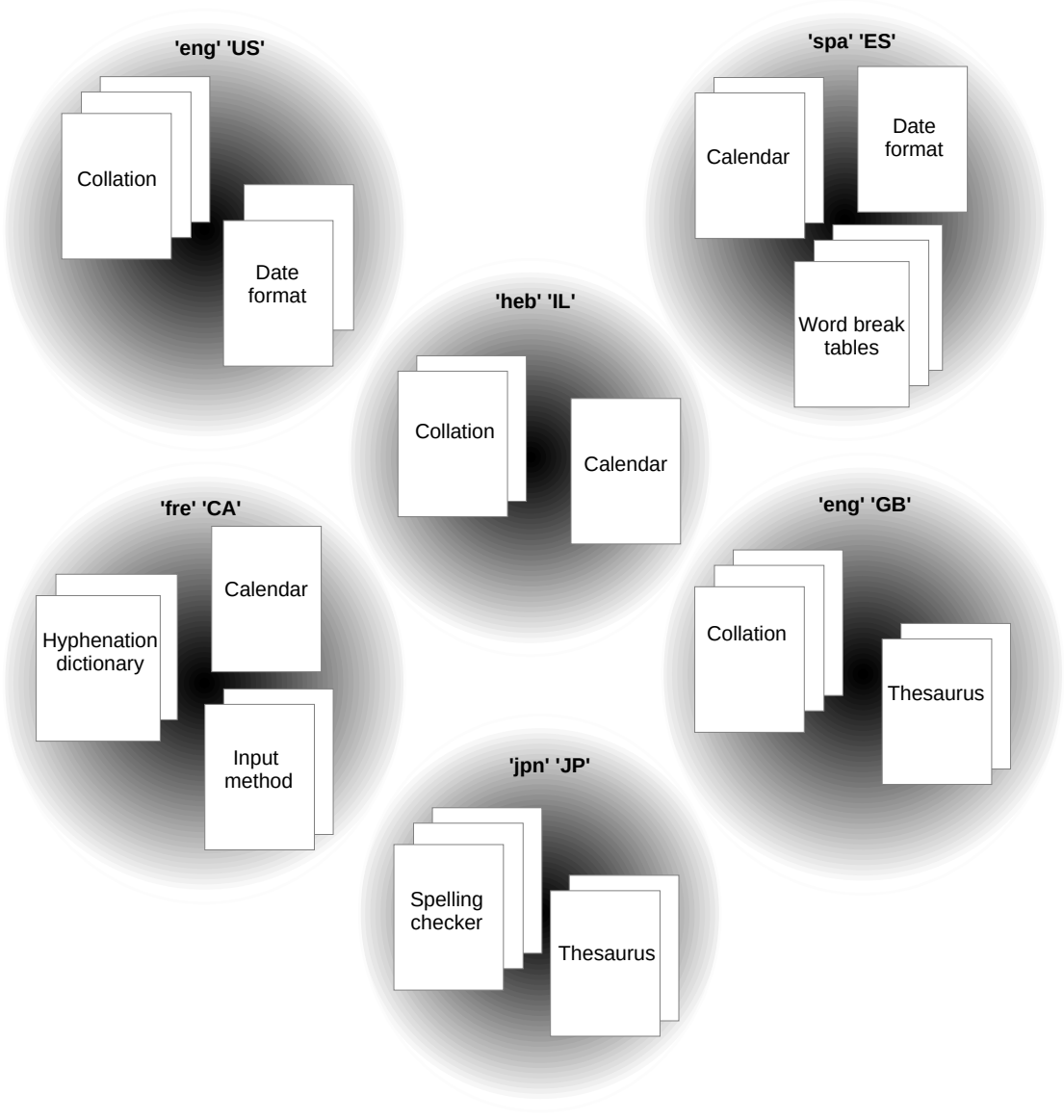
The Locale Database

The locale database is composed of clusters of information called **locales**, each of which pertains to a particular language and region. Each of these clusters is composed of various types of data used by text-handling operations and software plug-ins. International software components such as the Text Services Manager, the String Comparison Services, Date-and-Time Services, and other system components your application uses to process text use this data. Data stored in the locale database is also directly available to your application. All

Mac OS 8 applications use data stored in the locale database either indirectly or directly.

The locale database is composed of loose collections of data. It is highly extensible and flexible, accommodating various types of data. Using information accompanying the data that describes it, the Locale Object Manager catalogs this data in the database in a way that makes for easy access along multiple lines. Figure 1-11 shows a conceptual view of the locale database. (The contents of a locale database depend on information processed at system startup; see "How the Locale Database Is Created" (page 1-53).) Each locale is labeled with ISO language and region codes, together referred to as a **locale identifier**. These values define the primary language or geographical region of the locale and indicate the language and region to which the locale objects composing the locale belong. A locale identifier also contains a customization code identifying whether the locale contains customized data. Within each locale are shown the sets of locale objects composing it.

Figure 1-11 A conceptual view of the locale database



Locales

A locale exists to contain data for the language and geographical region that define it. Although the text encodings for different languages can share coded character sets, the languages for which they are used often differ in rules of composition. A region is a particular subset of a language. A region can represent a linguistic or cultural entity, not necessarily corresponding to a nation, whose language is different enough from other versions of the same language that it merits a specific localized version of Mac OS 8 system software. For example, U.S. and British are two regional variations that are subsets of the English language. The locale for the United States of America, for example, would have English as its default language and United States as its default geographical region.

A locale collects together locale objects containing data that establishes cultural preferences for the variation of the language used by a particular geographical region. The data belonging to a locale can specify a culture's text handling preferences for collation, word breaking, date-and-time formatting, hyphenation, and so forth. The data can also contain information providing access to input methods and other processes.

Although a locale contains locale objects for the culture represented by its primary language and region, a locale might also contain other kinds of locale objects. For example, a modern Greek locale might have locale objects containing collation tables or hyphenation dictionaries for classical Greek—perhaps one for Doric Greek and one for Attic Greek—for use by scholars of ancient Greek languages.

A locale also serves as a focal point in the locale database. The Locale Object Manager defines a locale reference data type that your application can obtain for any locale by specifying a locale identifier consisting of the locale's primary language and region.

Many application clients of the locale database want primarily to specify the default operation for various types of international processing. If your application is only interested in using defaults, it can find this information easily because locales cluster data for a specific language and region.

The Locale Object Manager

The Locale Object Manager provides a set of functions that manage, find, and provide access to data your application requires for international text processing and handling. Using the Locale Object Manager, you can search the

locale database for a single locale object containing data you want or search iteratively for any or all locale objects matching a set of criteria you specify.

You can use the Locale Object Manager for many purposes related to the task of finding and obtaining data. For example, you can

- determine the locale that is being used for your application, and change it if you like.
- obtain information about the database contents, such as the number of locales it contains and the default behavior for text-handling operations defined at system startup for any locale. You can also change these default behaviors for your application's use.
- find out the name and attributes of a locale object and the locale to which it belongs, in addition to obtaining that locale object's data.
- temporarily add objects to the database for your use and remove them.

For many of these processes, you must identify the locale, the locale object, or both where the data that you are interested in is stored or where the Locale Object Manager should begin looking for that data. For this purpose, the Locale Object Manager defines these two data structures:

- a locale reference that refers to one of the locales belonging to the locale database. You use a locale reference to specify the locale you are interested in when you call the Locale Object Manager functions to access and act on the data contained in locales, to specify the beginning position of a search, and to change the default locale to be used within your application's process. You can think of a locale reference as a resolved locale identifier that allows you direct access to a specific locale.
- a locale object reference that refers to a specific locale object. You pass a locale object reference to the Locale Object Manager functions that you use to obtain the data contained in a locale object or to obtain information about a locale object, such as any of the user-displayable names associated with the locale object, the locale object's key name, any of its attributes, and the locale to which it belongs. You can preserve a locale object reference and use it at any time to obtain a pointer to the data the object contains.

Default System Locale and Default Application Locale

At system startup, the Locale Object Manager establishes the default system locale based on the language and region for which the system is localized. The

default system locale identifies the locale whose content is used for international text-processing functions. Typically, the language of the system locale used for text-handling purposes corresponds to the language for which the system is localized. However, it is possible for these two to differ.

With the Locale Object Manager, you can obtain a reference to the default system locale without specifying its language and region. Once you have this reference, you can use it to determine the locale's language and region. The Locale Object Manager bases the default locale for your application on the default system locale. However, you can use the Locale Object Manager to change the locale for your application to one other than the system default.

How the Locale Database Is Created

At system startup, the Locale Object Manager builds the locale database from data contained in files stored in the Locales folder. The Locales folder can contain Locale files or any other type of file. A Locale file identifies the fundamental language or geographical region defining a locale and contains locale object resources of type `'lobj'` belonging to the locale. For each Locale file that the Locale Object Manager finds, it creates a unique locale in the locale database.

Other files stored in the Locales folder can contain stand-alone locale object resources. Locale objects are self-descriptive; they contain information specifying which language or geographical region they were primarily designed for. Based on this information, the Locale Object Manager associates each stand-alone locale object it finds with the most appropriate locale for it in the locale database.

Locale objects that are loaded into the locale database at system startup are considered permanently resident in the locale database—that is, your application, another application, or a system component that uses them cannot remove or permanently modify them, and they persist beyond the life of the application's process in which they are used. You can think of these locale objects as system resources.

Storing Persistent Data in the Locale Database

Your application can use the data that exists in the locale database without having ever stored any data there, and you can store data in the database. Most applications will use only the data stored in the locale database, but service

providers, localizers, and other groups of developers can provide data to be stored there.

Depending on your purpose, there are two approaches you can take to store data in the database and make it available to all applications and system components:

- You can define a locale and the locale objects composing it for the locale database by providing a Locale file.
- You can provide a stand-alone locale object, in any type of file, to be added to a locale defined by someone else.

Defining a Locale and Its Defaults

To define a locale, you provide a Locale file. It lets you identify not only the language and region to which the locale objects composing the locale belong but also the default behaviors for operations having many possible permutations.Within the Locale file, you provide locale object resources containing data used by these operations. You might provide a number of locale objects for the same operation. For example, your Locale file might contain several locale objects, each specifying a set of rules for string comparison for a given text encoding specification.

To identify which of these locale objects contains the default data, you can include a locale defaults list resource of type `'ldfl'` that specifies the default behavior for any given operation. In this way, you can characterize the default text-handling behavior of every operation within a locale. When no other information is available to determine which data is used, the data belonging to the default locale object for a particular operation is used.

Providing a Stand-Alone Locale Object

You do not need to define a locale to add a locale object to the database. You can provide a stand-alone locale object, and the Locale Object Manager will associate it with the most appropriate locale, based on required information you provide in the locale object.

When you install a file in the Locales folder containing a stand-alone locale object, you can easily include in the database any data that you want made available to all of its clients. You can just as easily remove this data from the database. For example, if you are a developer who intends to provide a specific utility or service, such as an input method, you can create and include a

stand-alone locale object in a file that you install in the Locale folder. When you want to supply a new version of the service, you can remove this file and replace it with an updated version to be used the next time the system is booted and the database is built. You don't need to include your locale object in a Locale file.

Locale Objects

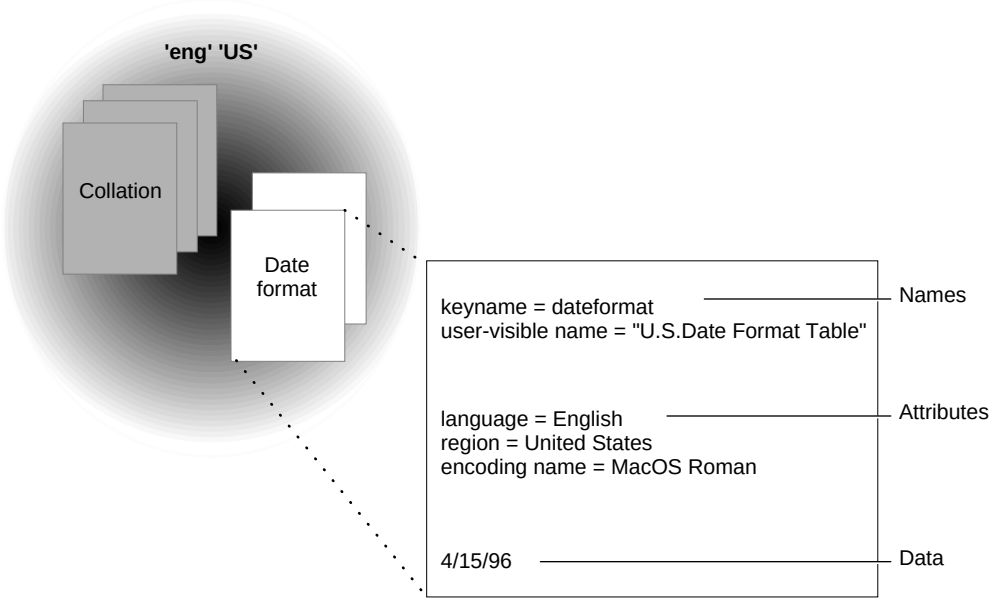
A locale object is an entity containing data localized for a specific text-handling operation or international software service, such as an input method or spell checker, and information describing that data. They can be incorporated in the locale database at system startup. They are installed in the database each time it is created and reside in the database permanently until the file containing them is removed from the Locales folder; if a file containing locale objects is removed, the next time the database is built these objects are not included in it. Unlike SOM-based text services—such as input methods and stemmers that use the Text Services Manager—locale objects incorporated in the database in this manner are data structures organized as resources; they are not objects as understood in terms of object-oriented design.

In addition to providing locale objects for use by any database clients, you can temporarily add them to the database for use by your application only. You use the Locale Object Manager for this; you don't need to create locale object resources for this purpose. Instead, you supply a pointer to data for the locale object when you call the function that adds it to the database. This is a void pointer, so you can provide any type of data for the locale object.

Regardless of the method you use to specify a locale object, you always provide additional information along with the text-handling data it contains. This information includes names and attributes that serve to catalog the locale object data, identifying the locale to which it belongs and the type of data it contains. This information makes the locale object accessible to you and other users of the locale database and also includes one or more text strings, telling about the data, that your application can display to your user.

Figure 1-12 shows a close-up view of a locale object containing an English U. S. date format table. Along with the data is a names table that contains the two required names and three attribute name-value pairs giving the language, region, and text encoding name, followed by the data. (See "Locale Object Attribute Name-Value Pairs" (page 1-57) for more information.)

Figure 1-12 Contents of a locale object



Note
A locale object resource contains additional information not described here that is used internally. ◆

The Locale Object Manager makes the culturally specific data contained in locale objects accessible to your application and other clients of the locale database from many different vantages. You can access the data by knowing only some aspects of it, for example, the kind of data it contains and the locale to which it belongs, or any of its attributes.

Locale Object Names Table

Every locale object in the database has associated with it a names table that contains at least two required name records. In addition to a key name, the names table always contains a user-visible name for the locale object. The **key name** is used internally to catalog the locale object in the database, and it serves as the primary search key. Two examples of key names are `inputmethod` and

`collatetable`. Like the key name, the user-visible name string indicates the type of data the locale object contains, only it is meant to be displayed to the user.

The optional names in the names table contain text strings that you can display to your user to describe the data contents of the locale object, for example, the copyright notice. You can obtain any of these name strings by calling the Locale Object Manager and specifying the name whose text string you want. Each locale object name has associated with it an identifier that serves to identify the type of data the name string contains. The Locale Object Manager defines constants for these identifier names that you can use to indicate the one you want. Here is a list of the name types that a names table can contain, along with their strings:

- The required locale object key name. The Locale Object Manager uses this name to catalog the data in the database. It also uses it as a key into the database to find locale objects of this type.
- The locale object user-visible name to display to the user.
- The copyright string name and the copyright value.
- The manufacturer string name and the manufacturer value.
- The function description name and a string that specifies the purpose or type of function provided by the object's data, for example, "U.S. English Collation Table".
- The locale object version string name and the version number value that gives the version of the locale object's data. For example, a version number value might specify the following string: "Apple Computer Japanese Input Method. Version 1.0".

Locale Object Attribute Name-Value Pairs

Every locale object in the locale database contains a set of attributes provided by the creator of the locale object, each of which consists of a name-value pair. Sets of attributes contained within a locale object serve to classify the data the object contains along multiple lines so that you can access it according to any collection of its qualities at different times. You provide a data structure containing attribute name-value pairs to describe the data that you are looking for when you call the Locale Object Manager to obtain it. For example, your application might look for all locale objects whose data is characterized by one

specific attribute while another application might look for locale objects of a certain type having in common two or more attributes.

Locale objects belonging to the same or different locales might have some of the same attributes; for example, all locale objects containing input methods and collation and number formatting services for a certain language, such as Thai, would have in common a language attribute for Thai.

Attribute name-value pairs distinguish locale objects having the same key name. Recall that a key name specifies the type of data a locale object contains. A locale itself can contain multiple locale objects that include data for the same type of operation. For example, the U. S. English-language locale might have two locale objects for date-format data: one showing numbers, one showing a mix of numbers and words. Taken as a whole, the locale database will contain many locale objects providing the same type of data.

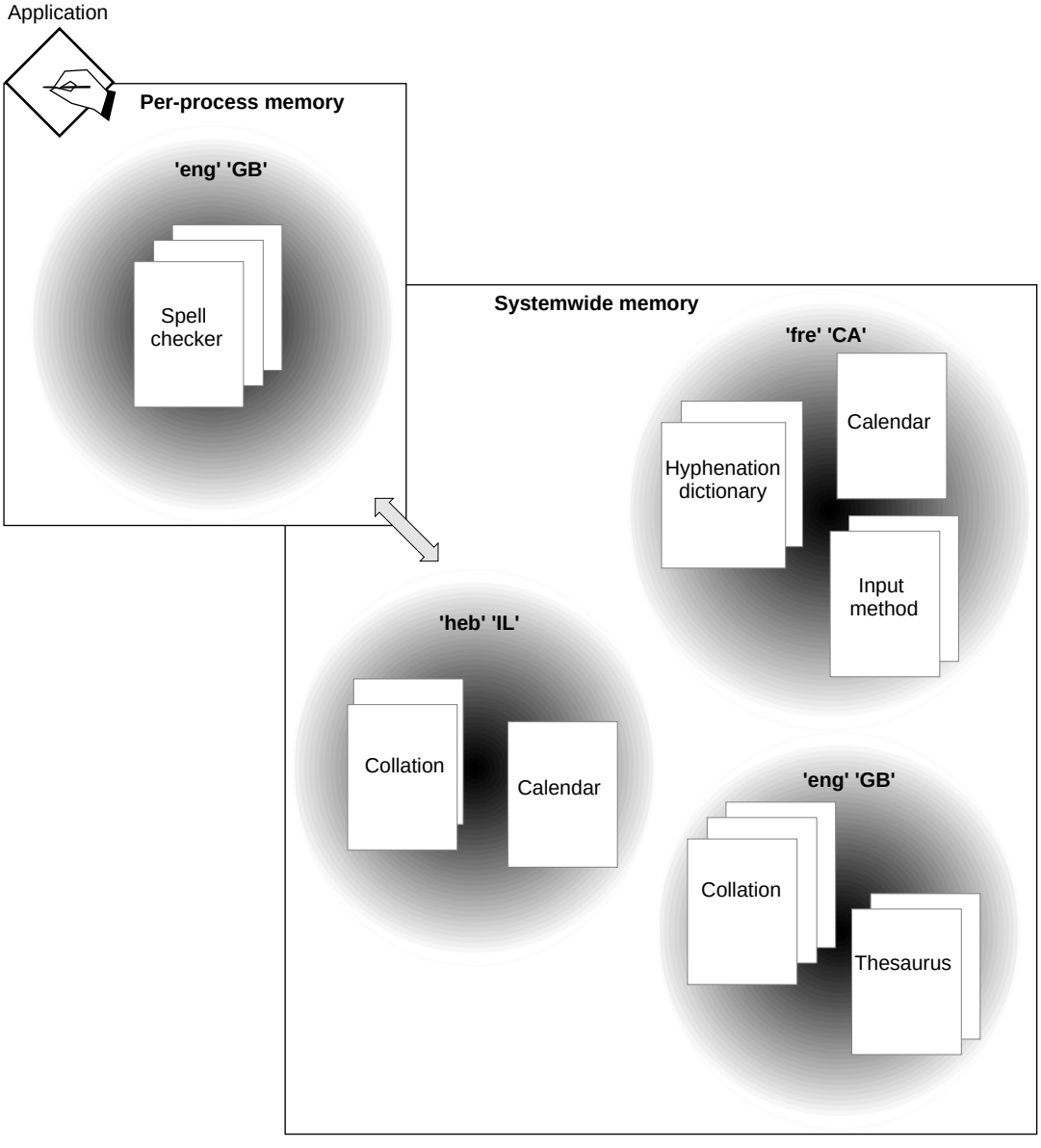
You can use attribute name-value pairs to specify which locale object of a certain type you are looking for. Suppose you want to obtain collation tables for the English language for both the British and U. S. geographical regions. To request this data, you would specify the key name and an attribute; for the key name you would specify `collatetable`; for the attribute you would specify the English language. If you did not qualify the key name with the language attribute, the Locale Object Manager would search the database for any locale objects containing collation table data without regard for the language they apply to. If you wanted only the regional collation table for the British form of the English language, you would further qualify your request by including another attribute specifying the region.

Recall that you provide an attribute value paired with a name. An attribute name describes the type of data the attribute value specifies. For example, a locale object might contain the predefined attribute name `kLanguageName` for which the associated attribute value is a specific language code. The Locale Object Manager defines a set of attribute names for commonly used attributes. You can ascribe these names to attribute values to identify their content type. These include constants for attribute names such as text service, keyboard input method, locale identifier, and SOM class. Associated with the text service name would be a value specifying a particular type of text service, and so on.

Where Locale Objects Reside in Memory

Locale objects composing the locale database can exist in system wide memory or in your application's per-process memory area, as illustrated by Figure 1-13.

Figure 1-13 Memory used for locale database

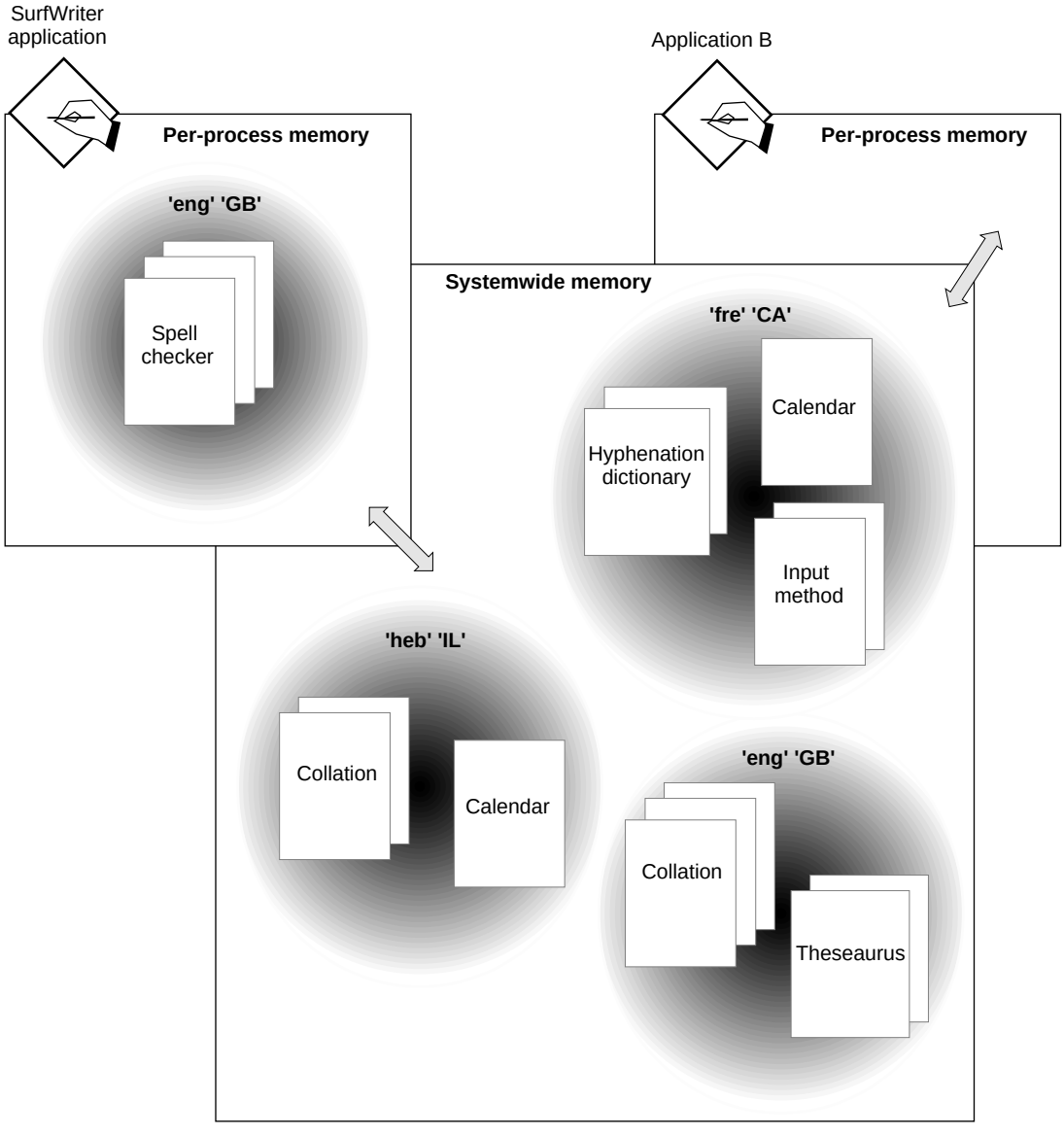


Locale objects that are loaded into the locale database at system startup are considered permanently resident in the locale database—your application cannot remove them or permanently modify them, and they persist in memory beyond the life of your application's process. These locale objects are stored in system wide memory. They essentially compose the locale database as it appears to all of its clients. Applications and system components using the locale database see this view of it.

However, you can use the Locale Object Manager to add a locale object to the database for use from within your application's current process. Any locale object that you add in this way is stored in your application's per-process memory area. From your application's view, the locale database appears to contain this locale object, but other applications accessing the locale database concurrently cannot see your additional locale object. When you add a locale object, you are not modifying the locale database, only extending its contents temporarily for your use.

Figure 1-14 shows where these locale objects reside in memory. It shows two views of the locale database. One view is from the perspective of the SurfWriter application that added a locale object to the database for use within its current process. This view includes the locale object stored in its per-process memory. Notice that the other application's view, Application B, is of the locale database in systemwide memory only.

Figure 1-14 Where locale objects reside in memory



Default Locale Objects for a Locale

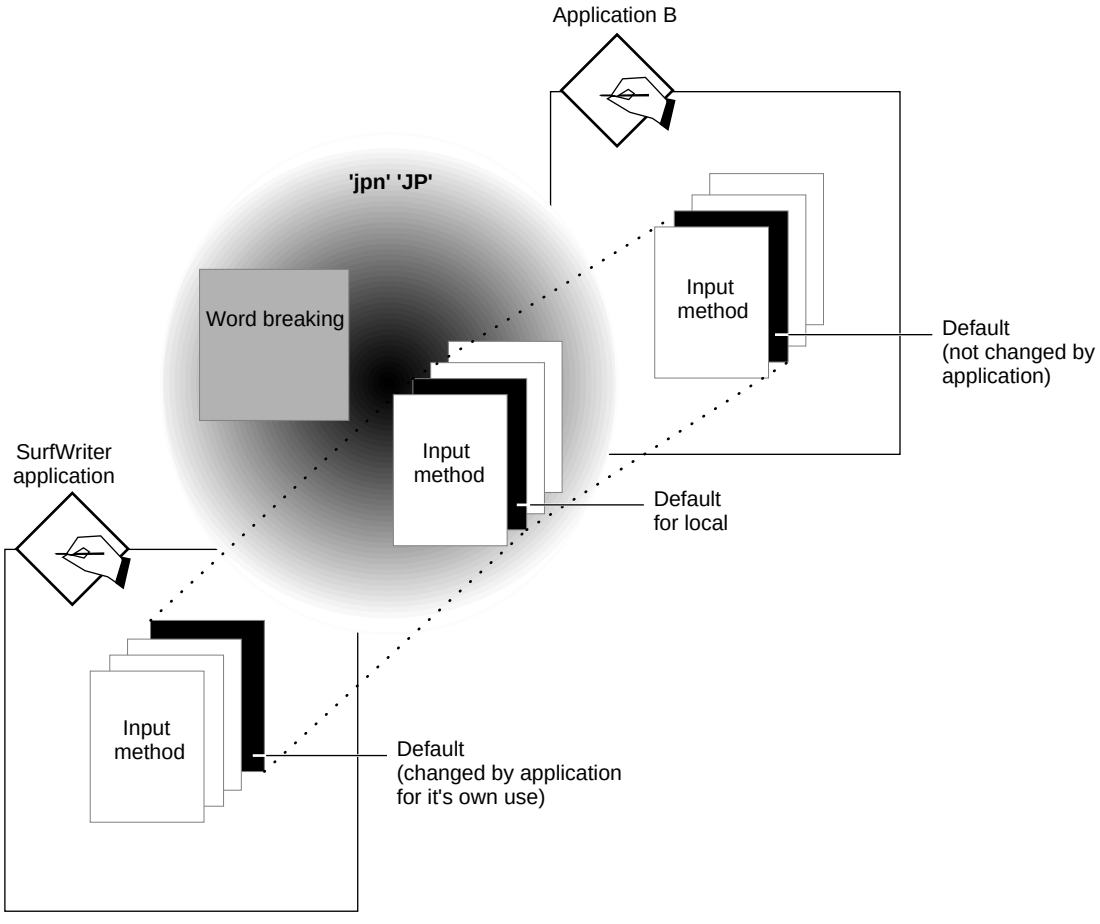
When the Locale Object Manager initially builds the locale database, it sets default behaviors for text-handling operations for each locale based on information provided in a Locale file. Your application can use the standard default behaviors for a given locale, or you can change one or more of them temporarily for use within your application's current process.

You can customize the formats of numbers, currency, time, dates, and measurements; you can customize string comparison, and other operations. For example, you might want to use short dates—the specification of dates in purely numeric representation. For the U. S. English locale, the short date for December 16, 1995 is 12/16/95. In this case, you would customize the U. S. English locale to use the locale object for this operation whose data contains the short-date format.

The Locale Object Manager provides functions you can use to obtain or change the data used to determine the default behavior for any text-handling operation that applies to a specific locale. To indicate the default behavior you want to know about, you specify the key name. To set the default behavior, you identify the locale object containing it. In both cases, you also identify the locale.

Any customizations you make to the default text-handling behaviors of a specific locale are valid for your application only from within its current process. Not only are these changes effective for your application only, but only your application perceives the database as modified in this way. Your modifications do not affect the locale database as it is seen by other applications that might be using it; the standard default values established at system startup are in effect for other applications accessing the database at the same time. Figure 1-15 illustrates this.

Figure 1-15 An application's view of default locale objects after changing one



Searching the Locale Database for Data

To obtain data stored in a locale object of the locale database, you call the Locale Object Manager, describing aspects of the data you want. The Locale Object Manager then searches the locale database for a match that satisfies the criteria you provide.

You can search for a single locale object, or you can search iteratively for more than one matching locale object. In either case, when the Locale Object

Manager finds a match, it returns a reference to the locale object and a pointer to the data it contains. You can use the pointer to access the locale object's data directly after calling the search function, or you can preserve the locale object reference and pointer and use it later to obtain the data, delaying retrieval of the data.

If you want to search for only one matching locale object, you can use the Locale Object Manager function designed for this purpose. You can position the search anywhere in the database by giving the locale reference to the locale where you want the Locale Object Manager to begin and by specifying the direction in which you want the search to proceed. The Locale Object Manager returns to you the first locale object it encounters that matches the key name and attributes you specify.

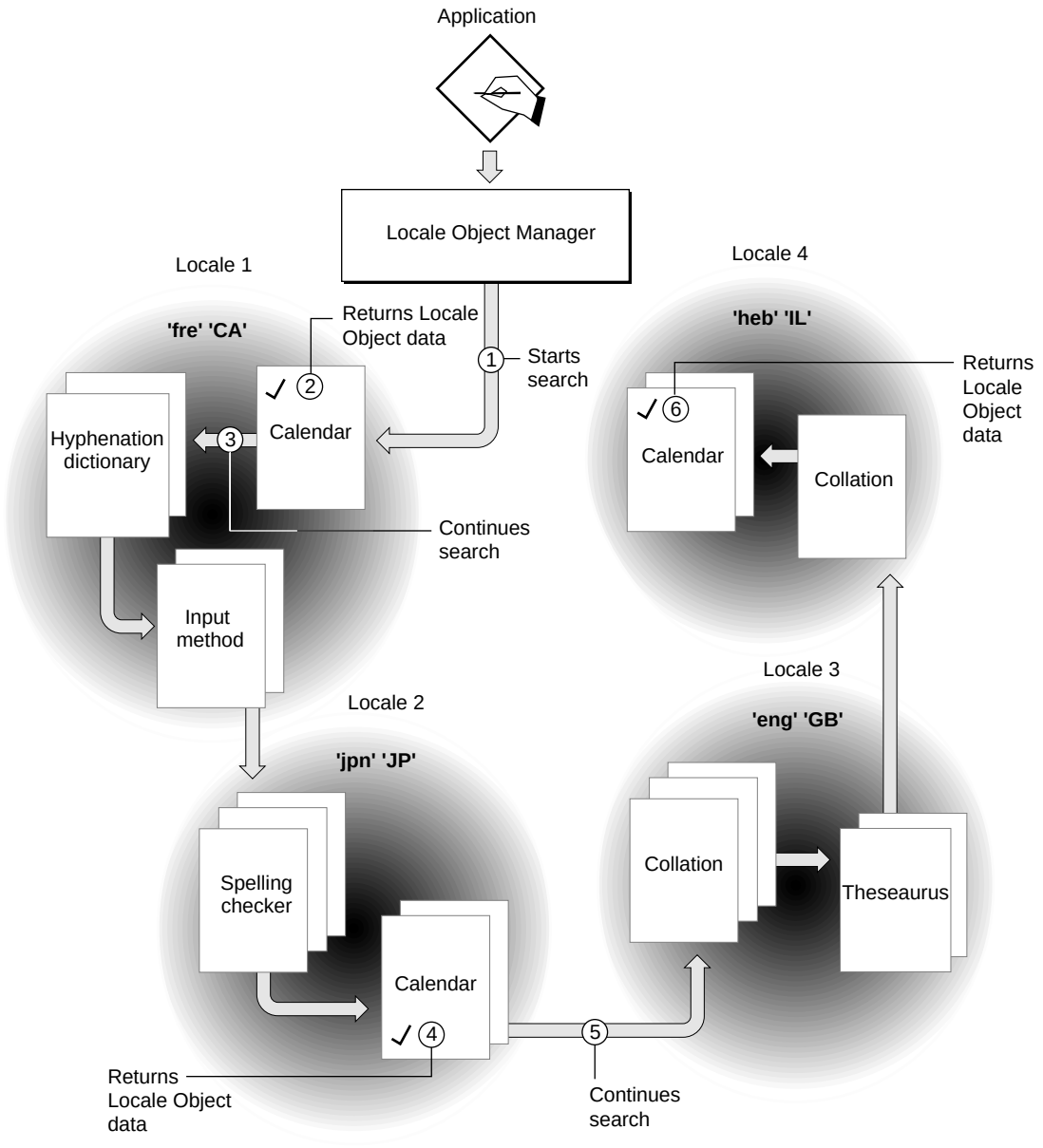
You can also perform an iterative search that traverses the entire locale database looking for matching locale objects; you use two Locale Object Manager functions designed for this purpose. To search iteratively throughout the database, you use a locale iterator. You create a locale iterator reference that contains a locale reference to the locale where you want the search to begin and the matching criteria, consisting of the key name and an array of attribute name-value pairs. You then pass the locale iterator reference to the Locale Object Manager function that performs the search, telling it the direction to proceed in. When the function encounters a locale object that matches your description, it returns a locale object reference and the data for the matching object to your application.

You can continue to call the function from within a loop to find all matching locale objects or until you find what you are looking for; you use the same locale iterator reference to do this. The Locale Object Manager tracks the progress through the locale database maintaining the next position at which to continue the search. When there are no more matching locale objects in the database, the function returns a result code that you can test against.

Figure 1-16 illustrates this process. In this example, the application is searching for locale objects containing calendrical data. The Locale Object Manager finds a match in locale 1— a matching locale object is marked with a check in this illustration. The Locale Object Manager returns the data to the application—indicated by the number 2, representing this part of the process. It then resumes the search at the point where it stopped—indicated by the process number 3—when the application calls the function again from within its loop. It doesn't find any more matches in locale 1, so it proceeds to locale 2. The Locale Object Manager finds another match in locale 2, returns it to the application—indicated by the process number 4. It again resumes the search

from where it left off—indicated by the process number 5 —after being called from within the loop. Notice that although it searches locale 3, the Locale Object Manager doesn't find a match there, so it continues on to locale 4, where again it finds a match and returns it to the application—indicated by the process number 6—completing the search of the entire database.

Figure 1-16 An iterative search of the locale database



Text Encoding and Conversion

If you are designing a text-intensive application, such as a word processor, your application should use the encoding converter services provided by Mac OS 8 when you want to convert the text from one encoding to another. This section describes the two types of encoding converters the Mac OS 8 provides and their uses, and it looks at some of the concepts underlying conversion between text encodings.

Note
In most cases, you should use text objects for text your application handles. If you use text objects for your multilingual applications, the system automatically handles any encoding conversions required by processes your application performs. The system software calls the encoding converter to convert the text from one encoding to another in a manner that is transparent to your application so that your code doesn't have to handle conversion. However, text objects are inefficient for use with the amount of text that applications such as a word processor handle. ◆

Encoding Converters

Mac OS 8 provides two encoding conversion managers—the High-Level Encoding Converter Manager and the Low-Level Encoding Converter Manager—that offer different levels of service for converting text across encodings. The high-level one is easy to use, having a simpler interface than the low-level one, but it gives you less control over the conversion process and provides less error reporting than does the low-level one.

The High-Level Encoding Converter Manager

The High-Level Encoding Converter Manager allows you to convert text between any two encodings. It does not map offsets pertaining to text formatting, as does the Low-Level Converter, so it is best used to convert mainly plain text or text with inline formatting, such as HTML.

The High-Level Encoding Converter Manager performs table lookup-based and algorithmic conversions. If the High-Level Encoding Converter Manager cannot perform an exact conversion, it takes the most reasonable action, effectively using default values. Although using the high-level version of the converter is easier and simpler than the low-level one, you cannot specify the conversion behavior, for example, for mapping strategy, through use of control options when you use it as you can with the low-level one.

The High-Level Encoding Converter Manager does not guarantee exact mapping. It may use a loose mapping or fallback-characters mapping in the conversion process. Because it does not provide detailed error reporting, your application is not informed when either of these types of mapping occurs. For this reason, the High-Level Encoding Converter Manager cannot ensure round-trip fidelity.

You should consider using the High-Level Encoding Converter Manager if

- you want to convert directly from any encoding to any encoding, but without exercising control over the process
- you're converting plain text or text with inline formatting
- you aren't concerned with high fidelity
- you don't need to control how mapping is performed when exact mapping isn't possible, and you don't need to know when loose mappings and fallbacks are used.
- you want to perform an algorithmic transformation

For table lookup-based conversions, the High-Level Encoding Converter Manager calls the Low-Level Encoding Converter Manager to perform the conversion. For conversions entailing text encoding schemes and conversions that perform algorithmic transformations for text encoding formats, it uses algorithmic conversion plug-ins. (See "Text Encoding Format" (page 1-76) for more information on transformations.)

The High-Level Encoding Converter is extensible, allowing you as a third-party developer to provide your own algorithmic conversion plug-ins to be used with it.

The Low-Level Encoding Converter Manager

The Low-Level Encoding Converter Manager performs table lookup-based conversions, allowing you to convert text encoded in the coded character set of

one text encoding to another using Unicode as a hub. (Table lookup-based conversion converts text encoded in a single text encoding to another; it does not deal with text encoding schemes.) The primary use of the Low-Level Encoding Converter Manager is to convert text from any text encoding to Unicode or to convert Unicode text to any text encoding. It does this efficiently and it gives you a fine level of control over how the conversion is performed. You can also use it to convert between any two text encodings when you want more control over the conversion mapping and extensive error reporting.

Through use of control options, the Low-Level Encoding Converter Manager allows you to specify how mapping should be performed. You can specify that you require high-fidelity mapping or you can stipulate the kind of mapping acceptable when a direct, exact, mapping is not possible; you can control whether the converter performs loose mapping and fallback mapping, and you can supply your own fallback handler for fallback mapping. These mapping concepts are discussed in "Converting Between Character Sets Using Mapping Tables" (page 1-79). The Low-Level Encoding Converter Manager can also map style or font information from a source text string to the converted string that it returns to your application so that you can maintain formatting information external to the text.

You should consider using the Low-Level Encoding Converter if

- you want to convert text in any text encoding to Unicode
- you want to convert Unicode text to any text encoding
- you require round-trip fidelity
- you want to convert text that has associated text formatting information and have the converter map the related offsets to the converted text
- you want control over how mapping is performed
- you want extensive error reporting when high-fidelity mapping isn't possible

The Low-Level Encoding Converter Manager does not perform algorithmic transformations; you should use the High-Level Encoding Converter Manager for this purpose.

Most applications will use the High-Level Encoding Converter for converting between any two text encodings. Applications that only want to convert either to Unicode or from Unicode will use the Low-Level Encoding Converter. However, when you want to convert text between any two text encodings and you require control over the process, you can call the Low-Level Encoding Converter Manager using Unicode as the intermediary encoding, or hub. Using

the Low-Level Encoding Converter Manager for this purpose entails a two-part process: first you convert text in any text encoding to Unicode, then you convert the text—now expressed in Unicode—from Unicode to the target text encoding.

The Low-Level Encoding Converter Manager consists of a set of functions and their data types, most of which align along these two processes: one group of functions allows you to convert text to Unicode; the other group allows you to convert text from Unicode. These functions also include truncation utilities that allow you to determine where to properly truncate text before converting it and utilities for converting Pascal strings to and from Unicode.

Characters, Codes, Text Encodings, Text Encoding Schemes, and Text Elements

In considering how text is converted from one encoding to another, it is useful to understand what constitutes a text encoding or a text encoding scheme. To understand what constitutes a text encoding, it is helpful to have a set of terms that describe its aspects and that clarify underlying concepts, which are often misconstrued. To use these concepts in a meaningful way that adapts itself to the practical and evolving requirements of text internationalization, it is important to make distinctions. Assigning terms to these concepts and defining the terms draws these distinctions.

Existing standards-setting bodies look at these concepts as they apply to internationalization from varying perspectives. For the most, they define their terms based on existing encoding schemes rooted in the past and the requirements these schemes met.

This section uses emerging terms and definitions for these concepts—characters, codes, coded characters, and text encoding schemes—suggested by proponents who are today creating the context in which internationalization is discussed and who are shaping its future.

Characters

A basic difference can be articulated in identifying how a person using a writing system might think of a character from how a computer handles one. The notion of a character exists in relation to writing systems; people usually think in terms of the graphical representation of a character. The encoding for a character exists in relation to computers; computers handle characters in the form of their numeric codes or representations.

A **character** is a unit of information used for the organization, control, or representation of data. Letters, ideographs, digits, and symbols in a writing system are all examples of characters. A character is associated with a name, and optionally, but commonly, with a representative image or rendering.

A **character repertoire** is a collection of *distinct* characters. Two characters are distinct if and only if they have distinct names in the context of an identified character repertoire. Two characters that are distinct in name may have identical images or renderings. Characters composing a character repertoire can belong to different scripts.

Codes

Computers do not recognize characters. Instead, they contend with their numeric representations or equivalents. Several terms come into play in thinking about how characters belonging to a character repertoire are represented to computers.

A **bit combination** is an ordered collection of bits that is interpreted as a binary number. A **code set** is a set of bit combinations of equal size that are ordered by their numeric values that must be consecutive. A **code set position** is the location of a bit combination in a code set. It corresponds to the numeric value of the bit combination.

Note
Some standards use the term **code point** to refer to a bit combination of a code set. The code point bit combination is the smallest unit of expression in a code set. All Unicode, Version 1.1 code points have a uniform width of 16 bits. ◆

Coded Characters

For characters to be recognized and distinguished from one another by computers, they must be mapped to code bit combinations at particular code set positions within a code set.

A **coded character set** is a one-to-one mapping from a character repertoire to a code set. These codes represent characters to the computer. A code set may contain bit combinations that do not correspond to a character in the character repertoire, that is, it can be a superset of the characters belonging to a character repertoire.

Note
Some standards bodies do not distinguish a coded character set from a character set and instead merge the concepts referred to by the terms character repertoire and coded character set in the single term *character set*. ◆

A **coded representation** is a sequence of one or more bit combinations that unambiguously represent a character in the domain of an identified coded character set. A code representation is often referred to as a coded character. A coded representation implies an object that is represented, specifically, a character. A given character may have more than one coded representation. Each distinct coded representation of a character is referred to as a "coded representation form."

A coded representation or element of a coded character set is a unit of encoding on which processing occurs. A coded representation should be independent of writing systems and textual data itself.

Text Encodings and Text Encoding Schemes

Computer users throughout the world who work mainly in their native language also need to use English characters or characters from other scripts. The Japanese writing system, for example, uses four individual scripts: Romaji (alphabetic Roman letters), Katakana and Hiragana (syllabic characters), and Kanji (ideographic characters). Even English-language users require use of pseudoscripts that include collections of symbols, numbers, and punctuation.

Providing support for multilingual applications entails finding a facile way to allow software to use characters from the scripts of multiple languages. This requires designing predictable ways to mix characters from multiple coded character sets.

A **text encoding** usually contains the encodings for the characters belonging to a single character set addressing a single script. To distinguish between a text encoding based in a single coded character set and a text encoding that addresses multiple coded character sets, editors of current Internet standards have suggested use of the second term *character encoding scheme* or *text encoding scheme*. Apple Computer aligns with this suggestion and uses the term *text encoding scheme*. A **text encoding scheme** is a method that specifies a unique mapping from a sequence of bit combinations to a sequence of integers, each of which is interpreted as the principal coded representation of a character in some identified coded character set. Text encoding schemes often include

predefined escape sequences that indicate transitions to specific coded character sets.

The sequence of bit combinations that serve as input to the mapping function of a text encoding scheme may be construed as a sequence of coded character representations. They may be preceded by, interspersed with, or followed by escape functions that must be explicitly specified by the text encoding scheme.

A well-known example of a text encoding scheme is the ISO 2022-JP-2 standard, which begins in ASCII and switches to other coded character sets of ISO 2022 through limited combinations of escape sequences. This text encoding scheme makes reference to 8 distinct coded character sets.

Many existing text encoding schemes are based on the method described in the ISO 2022 standard, which specifies code extension techniques. The ISO 2022 standard allows multiple coded characters sets to be combined in 7-bit or 8-bit formats; as mentioned previously, it identifies the various sets by escape sequences. The 7-bit format is used for software that cannot handle 8-bit data, such as some electronic mail programs.

Text Representation and Text Elements

When an application processes text, it usually decomposes the text into elements consisting of the smallest unit of data for a particular process. This fundamental unit of text is called a **text element**. Defining a text element is difficult because it is process dependent. One characteristic of a text element is that its definition changes depending on the operation in which it is used and even perhaps on the writing system to which it belongs. This makes it possible to think of a text element as an abstraction of a unit of text used by a particular process. Determining what constitutes a text element, then, depends on the particular language and process. For example, the Arabic ligature lam-alef text element is used for text rendering but not for text sorting.

No simple relationship exists between text elements and code representations. Coded character sets cannot enumerate all possible combinations of text elements, which are potentially unlimited. A single text element can correspond to a single code representation or multiple code representations. Multiple text elements can also correspond to a single or multiple code representations. Again, the relationship varies based on the language of the text and the application's process in which it is used. As mentioned earlier, a coded representation should be independent of writing systems and textual data itself. A code representation can be optimized for a particular text process, but this is not the same as making it dependent on that process.

Most text processing can be broadly classified into either of two categories: communication or computation. Communication involves the interchange of text. When an application transmits text, it sometimes need to convert the text from one representation to another in a predictable or algorithmic way to accommodate different media. This process entails modifying the form or content of the text.

Computation entails various processes. A common one is determining whether two text elements or sequences of text elements are equivalent. There are various levels of equivalency: are the text elements or sequences of text elements identical, do they take up the same code set positions, do they have the same form, do they render the same image when displayed, and so forth. Complex computational text processes include sorting, searching, text display, text editing, text word breaking, spell checking, and grammar analysis.

Different text processes operate on different units of text. For example, input processing must recognize natural boundaries that define text entered through the keyboard, pen, or voice.

A single process that is language dependent, such as sorting, uses different text elements, depending on the language of the text string, even when the process is applied to the same text. For example, the string “ch” is sorted differently for English and Spanish text. For English text, the string is treated as two text elements: “c” and “h.” For traditional Spanish text sorting, it is treated as one text element because it is sorted as a single character.

Because text element sequences depend on text processes, they may not always be ordered in the same way. Text element sequences can be ordered in different ways, for example, visually or logically. An example of a visual ordering is a display text element sequence for a given directional flow. An example of a logical ordering is one in which the code representations are ordered according to the input text element sequence.

Text Encoding Specification

For Mac OS 8, you use a data type called a text encoding specification to identify the coded character set, text encoding, or text encoding scheme in which a segment of text is represented. Mac OS 8 system components use text encodings and text encoding schemes indirectly or directly in handling text. You use the Low-Level Encoding Converter Manager to create a text encoding specification. Here are only a few of the ways in which text encoding specifications are used:

- Text objects contain the text encoding specification for the text they encapsulate.
- The Locale Object Manager locale objects contain text encoding specifications for user-displayable text strings they include.
- Encoding conversions performed by the either of the converters require at least two text encoding specifications. You identify the encoding of the source text and the target encoding to which you want that text converted.

A text encoding specification is an opaque scalar value into which the Low-Level Encoding Converter Manager packs four numeric values that identify the text encoding base, the text encoding variant, the text encoding format, and the packing version used for the text encoding or text encoding scheme.

When you create a text encoding specification, you specify all but the packing version. The Low-Level Encoding Converter Manager packs the three values that you provide into an unsigned 32-bit value, which you can then pass by value either directly or from within other data structures to the functions that use text encodings or text encoding schemes.

Text Encoding Base

A text encoding base is the primary specification of the text encoding or coded character set.

Text Encoding Variant

A text encoding variant identifies a text encoding or coded character set some of whose less commonly used characters vary from those specified by the base encoding scheme with which the variant is related. Variations in mapping usually exist only for insignificant characters.

Variants of the same base encoding usually coexist in the same system as font variants. Two different text encoding that can both be used for body text in the same language on the same version of a localized platform are considered variants of the same base encoding.

For example, the MacOS Icelandic and MacOS Turkish text encodings are considered different base encodings even though they belong to the same script; they normally do not coexist on the same Macintosh® system, and they each have their own language and region codes.

However, the Sai Mincho and Hon Mincho fonts, which are not distinguished by language or region, generally coexist on a MacOS Japanese system; they are considered variants of MacOS Japanese. Although Sai Mincho and Hon Mincho each implement slightly different character sets, they are not different enough for a user to think of them as something completely unique, as is the case with the Symbol and ITC Zapf Dingbats® fonts.

Text Encoding Format

A text encoding format identifies the packing format. It specifies the particular way in which a coded character set is algorithmically transformed, for example, to allow transmission through communication channels that may handle smaller bit values than those defined for the native coded character set, or to allow character codes to be handled by older software.

Typically, transformations are performed by programmatic code that implements an algorithm. For example, some communication systems require that data adhere to the rules of the ISO 2022 standard, which reserves the 8-bit code values between 0x80 and 0x9F (the C1 space), and the code position DELETE. Unicode uses these values to encode characters. As a result, direct transmission of Unicode data over these transmission systems is not possible.

Text encoded in a character set, such as Unicode, that uses 16-bit character encodings might be transformed programmatically by code that implements the algorithm of a specific format for transmission through a communication channel that handles 7-bit or 8-bit character codes.

The High-Level Encoding Converter Manager performs transformations using algorithmic conversion plug-ins. Third-party developers can install format transformation plug-ins for use with Mac OS 8.

Unicode

Most text encodings and text encoding schemes developed in the past offer limited or complex solutions to the problems intrinsic to text internationalization. Text encodings are limited, usually supporting one language, and text encoding schemes are characteristically complex. Although text encoding schemes can support a mix of encodings for processing groups of related languages or even collections of encodings for processing more unusual combinations of languages, such as a mix of Japanese and German, they entail convolutions such as use of escape sequences or reserved codes that signal shifts between encodings. A simpler solution would be to combine all code

representations of characters for all commonly used scripts and languages and symbols into a universal coded character set.

The Unicode coded character set provides this simple solution by attempting to encode all of the characters in use in the world today. It includes code representations for characters from the world's scripts as well as math operators, technical symbols, geometric shapes, and dingbats. Unicode uses a 16-bit encoding space, which its designers selected after carefully analyzing the overall requirements of the scripts which constitute modern written text. A plain text standard was defined to ensure legibility.

Unicode offers the simplest solution to problems inherent in providing support for fully multilingual systems. Any one text encoding scheme can provide support for all of the single encodings its method addresses. By addressing most of the character encodings for the world's scripts, Unicode can offer support for all common encodings and the languages they support. Because Unicode is a single coded character set, it doesn't require use of escape sequences or other complexities to identify transitions between coded character set. Unicode attempts to remedy problems common to application programs that handle multiple languages, such as use of multiple, inconsistent code representations caused by conflicting national character standards.

Using Unicode as the primary text encoding offers many advantages at the system level and to you as a developer of applications meant for the world market. Unicode provides more representational power than any other single text encoding scheme, enabling a vast diversity of languages to be expressed in one system. Because it encompasses code representations belonging to coded character sets used on most platforms and for most of the world's languages, Unicode facilitates data interchange with other platforms. Using Unicode, text manipulated by your application and shared across applications and platforms can be encoded in a single coded character set; this text can also be easily localized. Unicode offers advantages even to developers of English-only applications also because it contains a wide assortment of technical, typographic, and other symbols.

Unicode provides some special features, such as combining or nonspacing marks and conjoining jamos. These features are a function of the variety of languages that Unicode handles. If you have coded applications that handle text for the languages these features support, they should be familiar to you. If you have used a single coded character set such as ASCII almost exclusively, these features will be new to you.

The following two bodies, involved in the effort to standardize the world's languages for use in computing, define Unicode standards:

- The Unicode Consortium, a technical committee composed of representatives from many different companies, publishes the Unicode standard. The Unicode 1.1 standard is an evolution of the Unicode 1.0 standard, the first Unicode standard issued by the Unicode Consortium.
- ISO (the International Organization for Standardization) and the IEC (the International Electrotechnical Commission), national bodies that together form the specialized system for worldwide standardization, publish ISO/IEC 10646. This standard specifies the Universal Multiple-Octet Coded Character Set (UCS), a standard whose code point assignments are identical with Unicode.

The Unicode 1.1 Standard

The Unicode 1.1 Standard uses 16-bit character encodings. That is, all Unicode, version 1.1 code points have a uniform width of 16 bits. Unicode 1.1 is identical in code representation content to the ISO/IEC 10646-1 UCS-2 (Universal Character Set containing 2 bytes) BMP (Basic Multilingual Plane).

For this release of Mac OS 8, the Encoding Converter supports the Unicode Consortium's Unicode Standard, Version 1.1, specified by the *Unicode Standard: Worldwide Character Encoding, Version 1.1*. and the UCS-2 subset of the ISO/IEC 10646-1993 standard.

Note
Sixteen-bit character encodings have been proven sufficient to handle the world's commonly used written languages. However, the ISO/IEC 10646 standard includes a 32-bit encoding form referred to as UCS-4 (Universal Character Set containing 4 bytes). Although supporting the 32-bit encoding format is currently unnecessary because 16-bit encodings are adequate, in the future, Apple intends to support all Unicode encoding formats. ◆

ISO/IEC 10646

The ISO/IEC 10646 standard defines two alternative forms of encoding:

- a 32-bit encoding, which is the canonical form. The 32-bit form is referred to as UCS-4 (Universal Character Set containing 4 bytes)

- a 16-bit form that is referred to as UCS-2

The ISO/IEC 10646 nomenclature refers to coded characters as multiples of octets and assumes octets are serialized, while the Unicode nomenclature refers to coded characters as indivisible 16-bit entities. The ISO/IEC 10646 standard UCS-4 (32-bit character encoding) is not supported by the Unicode 1.1 standard.

Converting Between Character Sets Using Mapping Tables

To convert text between two text encodings, the Low-Level Encoding Converter maps the coded representations of characters from one set to another, taking into account complex conditions mentioned later in this section.

The Low-Level Encoding Converter Manager does not itself incorporate any knowledge of the specifics of any text encoding. Instead, it uses loadable, replaceable mapping tables that provide the information about any text encoding required to perform the conversion.

All information about a particular coded character set used in a text encoding is incorporated in a mapping table. A mapping table associates coded representations of characters belonging to one coded character set with their equivalent representations in another and accounts for the various conditions that arise when coded representations of characters cannot be directly mapped to each other.

A mapping table is stored as a resource file in the Text Encoding folder. One mapping table resource file exists for each supported base encoding.

Round-Trip Fidelity

When the Low-Level Encoding Converter Manager is able to convert a text string expressed in one text encoding to Unicode and back again to the original text encoding, with the final text string matching exactly the source text string—that is, without incurring any changes to the original—round-trip fidelity is said to have occurred.

For the various national and international standards that the Unicode Consortium used as sources for the Unicode 1.1 coded character set, Unicode provides round-trip fidelity. Because those coded character sets have been effectively incorporated into the Unicode coded character set, conversion involving them will always produce round-trip fidelity. Text in one of those coded character sets can be mapped to Unicode and back again with no loss of

information. Coded representations of characters that were distinct in the source encoding will be distinct in Unicode.

However, perfect round-trip conversion is not always possible. Not all vendor-provided coded character sets are directly incorporated into Unicode. Some code representations of characters may have no counterpart in Unicode. For example, a source text string from a vendor coded character set might contain a ligature that is not represented in Unicode. In this case, that information may be lost during the round trip.

The Low-Level Encoding Converter uses a variety of conventional methods to attempt to find some way to map the source coded representation of a character onto a sequence of Unicode coded representations in such a way as to preserve its identity and interchangeability.

Here are some of the methods used to map code representations of characters when high fidelity achieved through an exact or strict mapping is not possible:

- loose mapping
- fallback mapping
- mapping of characters to the Corporate Use Zone

Multiple Semantics and Multiple Representations

In many coded character sets, a single coded representation of a character may have multiple semantics, either by explicit definition, ambiguous definition, or established usage. A condition of **multiple semantics**, also called ambiguous semantics, exists when a single coded representation in one coded character set represents two distinct but similar text elements, and two separate coded representations exist for these text elements in Unicode.

For example, the JIS X0208 standard specifies the JIS X0208 character 0x2142 as having two meanings: double vertical line and parallel. Each meaning corresponds to a distinct Unicode code representation. The meaning “double vertical line” corresponds to the Unicode coded representation U+2016 “DOUBLE VERTICAL LINE”. The meaning “parallel” corresponds to the Unicode coded representation U+2225 “PARALLEL TO”. Either one is a valid match for the JIS character.

The ASCII coded representation 0x2D is specified as “hyphen, minus sign.” Unicode has a corresponding HYPHEN-MINUS character, which is the best match for the ASCII one. However, Unicode also has separate HYPEN and MINUS SIGN code representations of characters.

Multiple representation exists when a number of text elements can be represented in Unicode either as single coded representations or coded representation sequences. Examples include Latin and Greek coded representations with diacritics and Hangul syllables. The presentation forms encoded in Unicode can also be represented using coded representations for the abstract forms, and this also constitutes a condition of multiple representation.

Strict and Loose Mapping

A mapping table has both strict equivalence and loose mapping sections that identify how a mapping is to occur. Loose and strict mappings occur within the context of multiple semantics and multiple representations.

Strict mappings can be a one-to-one mapping (a mapping between one coded representation to one coded representation), a one-to-many mapping, or a many-to-one mapping.

In all these cases, strict mappings are exact mappings between coded representations yielding high fidelity when multiple possibilities exist. **Strict mapping** occurs when the mapping of a coded representation from Unicode to Character Set X, for example—a particular character set—is the exact reverse of the mapping of that coded representation from Character Set X to Unicode. **Loose mapping** occurs when the mapping of a coded representation from Unicode to the coded representation of a character belonging to another coded character set, for example, Character Set X, does not yield the same coded representation when the reverse mapping occurs, that is, when mapping from Character Set X to Unicode.

In the case of multiple semantics, a strict mapping exists between the single coded representation in Character Set X and only one of the two code representations in Unicode. Mapping to the other coded representation in Character Set X would constitute a loose mapping. Loose mappings from Unicode to Character Set X are considered additional mappings that match the semantics established for the coded representations in Character Set X.

Consider the example used earlier to illustrate multiple semantics: the single JIS X0208 coded representation 0x2142 that represents both of the characters double vertical line and parallel.

To map this coded representation to Unicode, it is necessary to choose between the two Unicode coded representations: U+2016 “DOUBLE VERTICAL LINE” or U+2225 “PARALLEL TO”. When this coded representation of a character is

mapped to either Unicode coded representation and then mapped back to JIS, the result is the same because both can be successfully mapped back to the single code representation. Round-trip fidelity has occurred.

However, this is not the case entirely when mapping from Unicode to JIS. If the Unicode coded representation U+2016 "DOUBLE VERTICAL LINE" is mapped to the JIS coded representation 0x2142 and back again, the end result—yielding U+2016 "DOUBLE VERTICAL LINE"—is identical to the starting coded representation.

But this is not true if the Unicode coded representation U+2225 "PARALLEL TO" is mapped to the JIS coded representation 0x2142 and back again; in this case, because the mapping is already defined, the end result is the Unicode coded representation U+2016 "DOUBLE VERTICAL LINE", which is not the starting character.

For the Low-Level Encoding Converter Manager functions that allow you to convert a single text segment or a text run from any text encoding to Unicode, you can set a flag specifying that the converter should use only the strict equivalence portion of the mapping table or that it can use the loose mapping section if the text element is not found in the strict equivalence portion of the table.

Presentation Forms

A **presentation form** is a graphic form that is used to represent a character when that character is used in a particular display format.

Presentation forms include variant forms, ligatures, and composite display forms. For example, presentation forms include some graphic symbols that represent multiple characters, such as those for Arabic contextual forms, Arabic ligatures, and Latin ligatures. Some coded character sets include different presentation forms for some CJK (Chinese, Japanese, Korean) punctuation and Japanese Kana characters depending on whether they are intended for horizontal or vertical display. Some coded character sets encode presentation forms instead of, or in addition to, encoding abstract characters. Presentation variants include full-width and half-width characters.

While text encodings designed before Unicode encode presentation forms, Unicode design goals specify that variant forms of characters that are predictable from the text content and context should not be encoded as code representations. For example, ligatures and composite display forms should not be encoded when they can be predicted from the text content and immediate context. Therefore, Unicode does not encode all presentation forms.

Corporate Use Zone

Code space in the Unicode standard is divided into areas and zones. One area, called the Private Use Area, includes a zone called the Corporate Use Zone.

Some code representations of characters that Apple requires are mapped to code representations in the Unicode Corporate Use Zone. The Apple logo is an example.

Apple provides a registry of its coded character set definitions in this zone that you can check to ensure that you don't use the same code representations. The path to this registry is `ftp:/unicode.org/pub/MappingTables/Apple`.

Although they allow the Low-Level Encoding Converter Manager to guarantee perfect round trips for certain code representations, characters in the Unicode Corporate Use Zone are not portable to other systems.

Fallback Mappings

A fallback mapping is a sequence of one or more bit combinations (or code points) in the target encoding for a text element that are not exactly equivalent to the source encoding bit combinations but which preserve some of the information of the original. For example, (C) is a possible fallback mapping for ©. In general, fallback characters are used as a last resort in converting text between encodings because they are not reversible and therefore do not lend themselves to round-trip fidelity conversions.

Fallback Handlers

A fallback handler is processing code that the Low-Level Encoding Converter Manager uses either when it cannot perform a one-to-one mapping in converting a Unicode character to another encoding or when it cannot use the strict mapping equivalence or the loose mapping portions of the specified mapping table for this purpose.

The Low-Level Encoding Converter Manager supplies a default fallback handler that you can associate with a data structure, called a conversion information reference, to be used for converting the text. However, you can also supply your own fallback handler and use it instead of or in addition to the default handler.

Base Encoding Mapping Tables Supported by Mac OS 8

Mapping tables that provide information about a particular text encoding exist for each supported base encoding. For Mac OS 8, the tables exist as resources in files that are stored in the Mapping Tables folder. (The folder name may have been localized.) There is one file for each base encoding that the converter supports. You can install your own mapping tables in this folder.

The Low-Level Encoding Converter Manager allows you to query it for a list of any mappings available on the system.

Handling Editable Text

Mac OS 8 minimizes the effort required of your application to handle editable text by introducing Text Editing Services, which consist of a text panel and a text engine, and their associated functions. Your application can call these functions to choose the text engine to use either with a text panel or alone. You also use these functions to insert and delete text, modify it, image it, and respond to user events related to text handling. TextEdit also belongs to Text Editing Services. However, to fully utilize Mac OS 8, your application should use text panels and text engines instead of using TextEdit directly.

Note
For this release of Mac OS 8, a modified version of the TextEdit engine that eliminates the 32K record limitation is the only supported text engine. ◆

When you use text engines directly, the interface is the same for any text engine. This consistency makes it possible and easy for you to use different text engines according to your text-processing and editing requirements. Because text engines are interchangeable with or without text panels, use of text panels and text engines allows for greater flexibility and extensibility.

The Text Panel

The text panel is a viewer through which your application user can enter and manipulate small amounts of text in fields and dialog boxes or in any window for which you support text input and editing. A text panel does not perform text processing or editing, nor is it aware of how this is done.

Text panels use text engines to perform most of their work. A text panel provides the user interface or the visible front-end portion and the text engine associated with it implements the text editing services such as text formatting, drawing, and editing.

A text panel references a text engine to be used with it. You can select the text engine that offers features you need from among any of the available text engines installed in the system. In Mac OS 8, you are not limited to use of a single text engine as with TextEdit in System 7. In addition to their special features, most text engines will offer standard text styling and attributes, such as plain text or boldfacing. Mac OS 8 provides text engines, and third-party developers can also provide text engines. Each text engine is registered with the system, along with a list of the features it implements.

Note
For this release, Mac OS 8 provides only the TextEdit text engine. This engine is a modified version of TextEdit that eliminates the TextEdit 32K record limitation. ◆

The text panel is a variation of the High-Level Toolbox (HLTB) panel. Its behavior differs from other types of panels only in the implementation of its methods dispatched due to events.

The Edit Text panel is provided for this release. The Edit Text panel is a specific implementation of the abstract text panel class. The Edit Text panel is designed as a class that defines not only user-interface behavior and application interaction— where and when the text is drawn—but also how it is drawn. It is tied to a specific text engine, the TextEdit text engine, for managing text.

Using the Text Panel

Text panels are simple to use requiring very little effort on the part of your application. For example, here's how you might display an editable text field in one of your application's windows. Once you have the signature of the engine you want to use, you call a Text Editing Services function to create a new instance of the text panel. You pass that function the ID of the text engine. You initialize the panel instance by also passing the function the rectangle for the text panel and the window it belongs to, and option bit flags, which allow you to specify if the field is editable, masked, off screen, and so forth.

That's all your application needs to do until it gets the text typed by the user to process it. For example, you no longer need to poll an event queue, as you

must in System 7, to determine which interface element is concerned with an event. The text panel manages itself in the rectangle you defined within the window. After you instantiate and initialize the text panel, your application can stay out of the way (through `AEReceive`) until one of its service routine is invoked by the system. At that point, your application can resume activity to implement the functionality of a user-interface element instead of managing the entire process.

The Edit Text panel uses the default system handlers to intercept events and pass them to an associated text engine for processing. This occurs transparently, requiring no effort on the part of your application.

You can use the Text Editing Services functions to determine whether the text is read only, selectable, or maskable. You can set and get the bounds where the text flows, that is, the bounding height and width. You can install, extract, delete, and replace the text edited by the text engine. You can style the text. Standard text styling features, such as bold, italic, underline, and outline are supported. To determine whether other styles are supported, you can query the text engine. You can set and get the text color and alignment, and you can get the text font and size. You can count the number of faces used for a given range of text. You can highlight, select, or draw a specified range of text. These services also include a set of text attribute iterator functions that you can use to respond to a user's actions that modify the style attributes of the text.

Text Engines

You use a text engine with a text panel, but you can also use a text engine alone to perform any task you want.

The Mac OS 8 design for Text Editing Services separates the text engine definition from the text panel to make it possible for you to use any available text engine you want. This means that although you can still use TextEdit as your text-editing engine, you are no longer dependent on it. The default system engine for Mac OS 8 is based on TextEdit to preserve backward compatibility with the Dialog Manager text fields in System 7, but new text engines will be made available from Apple Computer and third-party developers. The system is designed to allow you to choose a text engine that offers the services which best accommodate your text-editing requirements.

All available text engines are registered with the system. When a new text engine is installed, for this release of Mac OS 8, it is registered in the locale database, with its specific features and a text engine signature. (Note that use of

the locale database is subject to change in the future.) Your application can ask for a text engine that meets your requirements by querying the locale database for a specific attribute. For example, you can ask for one that supports GX typography, large text, or one that supports WorldScript®. In return, you'll get the signature of a text engine. Your application uses this signature to load the corresponding SOM object either directly or through the text panel.

Using a text engine directly gives your application more control and capabilities than you have in using a text panel. When you use a text engine with a text panel, the text panel inherits and overrides all event-handling routines that are routed automatically from the window to the panel. When you use a text engine directly, your application should explicitly call the text engine event handlers.

All text engines use the same group of data structures and methods required to provide core text-processing and text-editing services, that is, they all have the same API. This means that you do not need to know how to deal with the particulars of different text engines.

Selecting and Getting a Text Engine

Whether you use a text engine with a text panel or directly, you need to identify the text engine you want. The Text Editing Services provide three separate functions that allow you to get a text engine based on different information. Here are the three ways in which you can do this:

- You can get a text engine by signature. Each text engine has a unique signature.
- You can get a text engine by class name. Each text engine has a unique descriptor consisting of its SOM class name and SOM minor and major version number.
- You can get the default engine.

If you are interested in a specific kind of text engine offering certain features, you can use a function that finds a matching text engine, if one is registered with the system, based on a list of features you pass to the function.

Once you have a text engine, you can call a different function to query whether that text engine has a specific feature you are interested in before you create an instance of the text engine. For example, you might want to know if the engine supports strike-through or double underline. The Text Editing Services provides an enumerated list of predefined features that you can check for.

Using a Text Engine Directly

To use a text engine directly instead of with a text panel, you initialize and activate the text engine instance after you get its reference. You can choose the level of control you want over the engine and determine how the engine reacts to certain events. You can direct the engine to install its own handlers, or not, when it is activated. This is not possible when you use a text engine in conjunction with the text panel, because the text panel implementation determines how events are handled.

You can use a text engine directly, for example, if your application supplies a simple text editor, and allow the system default handlers to intercept Apple events and communicate with the text engine. Alternatively, if you need to intercede, for example, to filter events to restrict the kind of data the user enters, you can override some of the system's default event handlers by calling Apple Event Manager functions to stack your own handler table above the system default handler table of the Apple event dispatcher for your process.

The Text Editing Services includes functions for initializing a text engine, specifying the width and height of the rectangle where the text should be formatted, setting and getting the margins added to the text frame rectangle, drawing text, inserting, deleting, and replacing text, getting and setting text attributes, imaging text according to page dimensions for printing, enabling and disabling text drawing for special purposes such as search-and-replace, selecting (highlighting) ranges of text, and scrolling the text.

They also include functions for storage and scrap management, and for manipulating raw text in order to provide services for word-breaking or text-element breaking.

Drag-and-drop is completely supported and implemented by the engine without requiring any intervention on your application's part. However, if for any reason you want to handle mouse-down events yourself, the Text Editing Services provide functions that enable this for drag-and-drop support.

If you direct the text engine to install its own handlers, you don't need to be concerned with mouse-event handling. However, if you want to handle mouse events yourself, the Text Editing Services provide functions for this purpose.

If You Are Providing a Text Engine

The structure of a text engine is specified in SOM interface definition language (IDL). Mac OS 8 defines the default engine TSystemTextEngine. You can override the default engine and provide your own. However, you should do

this only to provide cosmetic changes. Because the data structure of a text engine is opaque, you will not be able to change radically the behavior of an existing engine. Instead, if you want to change the behavior, you should provide your own engine.

If you implement a specialized text engine, whether its for your application or for others, you must make the engine available. To make your text engine available, you must register it with a central service and provide a list of attributes that describe your text engine's features. Information on how to do this will be provided with a later developer release.

About TextEdit

Other parts of the Text Editing Services do not replace TextEdit; they offer a higher-level service that is easy to use and that lets you to use one of the available text engines or your own text engine instead of limiting you to TextEdit. Although you can still use TextEdit directly, Apple recommends that you use text panels and text engines instead of TextEdit for your Mac OS 8 application.

Text Editing Services include an enhanced version of TextEdit that includes integrated inline input support, integration of drag-and-drop support, and support for text objects. It is based in the new event model.

String Comparison

Mac OS 8 provides the String Comparison Services, a set of functions for comparing and searching text objects and strings. These functions provide many improvements over the System 7 string comparison routines. They allow for portability, better performance, better linguistic capability, the ability to handle Unicode, and easier localization.

Collation References

The String Comparison Services include a private data structure called a collation reference (`CollationRef`) that collects all information relevant to a particular, desired collation order, including references to appropriate locale objects and any default overrides your application uses. The String

Comparison Services allow you to create, modify, and destroy collation references. However, you do not need to provide a collation reference when you use collation functions. If you don't provide one, the function uses the default behavior for the locale of the current process.

Overriding Default Collation Behavior

Some developers want a high degree of control over aspects of collation behavior. Usually, these developers do not want to change the collation behavior for the letters of a particular language, but they want to be able to change the relative order of scripts and how punctuation and numbers are handled, for example. For those of you who want this kind of control, the String Comparison Services provide functions that allow you to override the default behavior for aspects of collation.

You can use the `OverrideCollationSetOrder` function to override the relative order of character groups and classes defined by the locale. A *group* is a logical collection of characters that are related to a script or common to several scripts; these characters may include letters, punctuation, numbers, and so forth. For example, all the characters that are specific to Greek constitute a group. Some groups include characters that are common to several scripts: the general punctuation, numbers, and symbols in Unicode are common to all scripts. Localizers generally define the collation order for one or more groups.

Code Conversion for String Comparison

You can collate text objects and text strings. If the text objects and text strings to be compared are in different encodings, the String Comparison Services will call the Low-Level Encoding Converter to convert both of them to Unicode before comparing them. All locales of the locale database should contain collation tables for Unicode, even if they don't have them for other encodings, which makes this collation feasible. If the text objects or text strings are in the same encoding but no collation tables exist for that encoding, they will also be converted to Unicode if Unicode collation tables exist.

The String Comparison Services can handle text strings that use the same or a mixture of encodings. For cases of multiple encodings that force a code conversion, the String Comparison Services automatically call the Encoding Converter in a manner that is transparent to your application.

Here are the four possible types of encoding cases that the String Comparison Services handle:

- *Unicode and Unicode, case A.* In this case, conversion of either of the two text strings is not necessary. If collation tables for the Unicode encoding are missing the required ranges, the String Comparison Services return an error.
- *Unicode and any other encoding (referred to as Character Set X), case B.* In this case, the text string in Character Set X must be converted to Unicode before the String Comparison Services can compare the two text strings. If the String Comparison Services cannot do this, because, for example, conversion tables are missing, then it returns an error. Otherwise, it performs a Unicode-to-Unicode comparison on the strings.
- *Character Set X and Character Set X, case C*. This case—most likely the most common case for early releases of the Mac OS—may or may not entail conversion. First, the String Comparison Services check to see if collation tables for the text strings are present. If so, it uses them. If not, it tries to convert the text string in Character Set X to Unicode and do comparison using collation tables for Unicode; this process, then, becomes the same the as Case B, described above.
- *Character Set X and another, different, non-Unicode encoding (referred to as Character Set Y), case D*. In this case, both Character Set X and Character Set Y must be converted to Unicode before the String Comparison Services can perform comparison. If either comparison fails (due to missing tables or something else), the String Comparison Services return an error. If both text strings are successfully converted, the String Comparison Services perform a Unicode-to-Unicode (Case A) comparison on them.

Locale Object Manager Reference

Contents

Locale Object Manager Constants and Data Types

The Locale Object Manager provides a set of functions that manage, find, and provide access to data required by international system components and applications. These data are stored in the locale database. The database serves as a repository of international preferences and data organized into sets of information clustered along cultural lines, each of which composes a **locale**. Each locale represents a particular cultural entity. Locales are composed of **locale objects** that contain data pertaining to the locale's culture. In addition to data, locale objects contain names and attributes describing their data and its use in various ways. Your application uses these names and attributes to identify the content of locale objects whose data you are interested in. A locale can, and usually does, have multiple locale objects.

Locale Reference

A **locale reference** is a private data type that refers to one of the locales belonging to the locale database. You use a locale reference to specify the locale you are interested in when you call the Locale Object Manager functions to access and act on the data contained in locales and to change the default locale to be used within your application's CFM context. You can think of a locale reference as a resolved locale identifier (page 2-16).

You can search the **locale database** for information beginning from any locale. You use a locale reference to indicate a specific locale where you want to begin a search of the database. You can use the `GetLocaleReference` function (page 2-25) to obtain a locale reference for a specific locale. You specify the locale for which you want a reference by giving its local identifier (page 2-16).

At system startup, the Locale Object Manager establishes the default system locale based on the language and region for which the system is localized. The default system locale identifies the locale whose content is used for international text processing functions. Normally the system locale corresponds to the language the system is localized for. However, it is possible for the default system locale to differ from the language for which the system is localized.

You can use the `GetSystemDefaultLocaleRef` function (page 2-25) to obtain a reference to the default system locale.

The default system locale becomes the default locale for your application. However, you can change the application locale by calling the `SetCurrentProcessLocale` function (page 2-26). At any time, you can obtain a reference for the locale of the current process by calling the `GetCurrentProcessLocaleRef` function (page 2-22).

A locale reference is defined by a `LocaleRef` data type.

```
typedef struct OpaqueLocaleRef* LocaleRef; /* locale reference */
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Locale Iterator Reference

When you need to obtain data that pertains to a specific culture or certain types of data for processing text for different cultures—for sorting or word breaking, for example—you can query the locale database; the locale database contains the data or it contains information giving access to culturally related data stored elsewhere.

You can locate and retrieve certain types of data associated with different locales contained in the locale database by specifying a **locale object key name**. Locale object key names contribute to the information that the Locale Object Manager uses to catalog and find locale objects in the database.

You can refine the search for locale objects by specifying one or more locale object attributes (page 2-8), for example, a specific language for which you want input methods. The locale object name and attributes you specify are used to search the database for locale objects containing matching values.

To search iteratively for more than one locale object that satisfies your matching criteria, you must first create a **locale iterator reference** containing information used to perform the search, such as where the Locale Object Manager should begin the search and the matching criteria it should use. A locale iterator reference is a private data structure created and maintained by the Locale Object Manager and returned to your application when you use `LocaleIteratorCreate` (page 2-42) to create one.

You pass the returned locale iterator reference to `LocaleIterate` (page 2-47) to search iteratively through the locale database looking for locale objects whose locale object name and attributes match those you specified. A locale iterator reference is defined by a `LocaleIteratorReference` data type.

```
typedef struct OpaqueLocaleIteratorReference* LocaleIteratorReference;
    /* locale iterator reference */
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Locale Database Search Direction

You can specify the direction in which you want the locale database search to proceed—forward or backward from the starting locale position—when you use `LocaleIterate` (page 2-47) to perform an iterative search throughout the database. You can use the constants defined by the following enumeration to specify the direction:

```
typedef UInt16 LocaleIterateOp;
enum {
    kLocaleForwardIterate   =   0,  /* foward databse search */
    kLocaleBackwardIterate  =   1   /* backward databse search */
};
```

Enumerator descriptions

`kLocaleForwardIterate`
: Search forward in the locale database starting from the locale specified in the locale iterator reference (page 2-6).

`kLocaleBackwardIterate`
: Search backward in the locale database starting from the locale specified in the locale iterator reference.

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Locale Object Reference

A **locale object reference** is a private data structure that refers to a specific locale object. You pass a locale object reference to the Locale Object Manager functions that you use to obtain the data contained in a locale object or to obtain information about a locale object, such as any of the user-displayable names associated with the locale object, the locale object's key name, any of its attributes, or the locale to which it belongs. You can preserve a locale object reference and use it at any time to obtain a pointer to the data the object contains.

The Locale Object Manager returns a locale object reference when you search the locale database for a locale object and when you temporarily add a locale object to it. These three functions return a locale object reference:

- When you use `SearchOneLocaleObject` (page 2-40) to find the first locale object that matches your search criteria, the function returns a locale object reference, identifying the locale object, along with its data.
- For each matching locale object it finds, `LocaleIterate` (page 2-47) returns the locale object reference of the locale object that satisfies the search-matching criteria along with that locale object's data.
- When you use `AddLocaleObject` (page 2-51) to add a locale object to the database for the duration of the current process, it returns a locale object reference to you.

A locale object reference is defined by a `LocaleObjectRef` data type.

```
typedef struct opaque LocaleObjectRef; /* locale object reference */
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for more information.

Attribute Name-Value Pair Structure

Every locale object in the locale database contains an arbitrary set of attributes, each of which consists of a name-value pair. Sets of attributes contained within a locale object serve to classify the data the object contains along multiple lines so that it can be accessed according to any collection of its qualities at different times. For example, your application might look for all locale objects

containing one specific attribute while another application might look for locale objects having in common two or more attributes.

An attribute name describes the type of data the attribute value specifies. For example, a locale object might contain the predefined attribute name `kLanguageName` for which the associated attribute value is a specific language code.

You can provide one or more attribute name-value pairs as matching criteria to be used to search the locale database for locale objects. Along with a locale object key name, attribute name-value pairs identify and distinguish two locale objects that have the same key name. You can retrieve data contained in one or more locale objects belonging to the same or different locales by specifying a set of attributes; all locale objects having those attributes are made available to your application.

You specify attribute name-value pairs for use with the `SearchOneLocaleObject` function (page 2-40) to search for a single locale object—the function returns the first locale object found that satisfies the matching criteria. You specify attribute name-value pairs when you create a locale iterator reference (page 2-6) for use with the `LocaleIterate` function (page 2-47) to search within and across locales for locale objects that satisfy the matching criteria.

To obtain all attribute name-value pairs associated with a specific object, you use the `GetLocaleObjectAttributes` function (page 2-31).

You use an attribute name-value pair structure to specify an attribute. To specify more than one attribute, you pass the function a pointer to an array of name-value pairs. The attribute name-value pair structure is defined by the `NameValuePair` data type.

```
struct NameValuePair {
    StringPtr   name;           /* name for attribute value */
    ByteCount   valueLength;    /* length of attribute data */
    void        *value;         /* attribute data */
};
typedef NameValuePair *NameValuePair;
```

Field descriptions

`name` A Pascal string that ascribes a name to the attribute value. The name describes the type of data the value carries. For this parameter, you can use one of the constants for the

	attribute name strings (page 2-10) defined by the Locale Object Manager for common attributes.
`valueLength`	The length in bytes of the attribute data given in the `value` parameter.
`value`	The attribute data.

See "Locale Object Manager Constants and Data Types" (page 2-5) for more information.

Standard Attribute Names

You use attribute name-value pairs (page 2-8) as part of the search criteria you give when you use the Locale Object Manager to find data in the locale database.

A **locale object attribute** consists of a name-value pair that serves to classify the data a locale object contains in a particular way. A locale object includes an attributes table that contains various attributes, allowing the locale object to be categorized along multiple lines so that you can access it according to any collection of its qualities at different times.

The Locale Object Manager defines a set of attribute names for commonly used attributes. You can ascribe these names to attribute values to identify their content type. A locale object may have associated with it many other types of attributes each of which has a name consisting of an ASCII string not defined by the Locale Object Manager.

You can use these constants defined for the Locale Object Manager's attribute name strings as the value of the `name` field portion of an attribute name-value pair structure. The Locale Object Manager defines these standard attribute names:

```
#define kExecutableCfragName    "\pexecutablecfrag"
    /* attribute contains name of executable code fragment */
#define kSOMClassName           "\psomclass"
    /* attribute contains SOM class name */
#define kUserVisibleName        "\puservisiblename"
    /* attribute contains user-displayable name */
#define kScriptName             "\pscript"
    /* attribute identifies locale object data as System 7 international
        resource */
#define kLanguageName           "\planguage"
```

```
    /* attribute identifies language locale object data applies to */
#define kRegionName             "\pregion"
    /* attribute identifies region locale object data applies to */
#define kResIDName              "\presid"
    /* attribute identifies locale object data as System 7 international
        resource */
#define kEncodingName           "\pencoding"
    /* attribute specifies text encoding */
#define kTextServiceTypeName    "\ptextservice"
    /* attribute identifies the text service */
#define kInputMethodTypeName    "\pkeyboardinputmethod"
    /* attribute specifies the type of input method */
#define kLocaleIdentifierName   "\localeidentifier"
    /* attribute specifies the locale identifier */
```

Constants descriptions

`kExecutableCfragName`
: Specifies that the attribute value contains the name of the executable code fragment.

`kSOMClassName`
: Specifies that the attribute value contains the SOM class name.

`kUserVisibleName`
: Labels the attribute value as containing text intended to be displayed to the user, such as copyright information.

`kScriptName`
: Marks data, identifying it as formerly System 7 international resource type data.

`kLanguageName`
: Labels the attribute value as specifying the language to which the data contained in the locale object applies.

`kRegionName`
: Labels the attribute value as specifying the region to which the data contained in the locale object applies.

`kResIDName`
: Marks data, identifying it as formerly System 7 international resource type data.

`kEncodingName`
: Labels the attribute value as specifying the text encoding. See the "Low-Level Encoding Converter Manager Reference" for a description of text encodings.

`kTextServiceTypeName`
: Specifies that the attribute value identifies the function provided by the text service.

`kInputMethodTypeName`

Labels the attribute value as specifying the input method type.

`kLocaleIdentifierName`

Labels the attribute value as containing a locale identifier value.

Any client of the Locale Object Manager—including other system components—can define name strings that are appropriate for the data contained in locale objects stored in the database.

If you are interested in a set of standard name strings for a particular system component, such as the Text Services Manager, refer to the reference chapter for that component.

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Name-Table Entry

Every locale object in the database has associated with it a **locale object names table** that contains at least two name records for the set of names associated with the locale object. In addition to the locale object key name, the names table always contains a user-visible name for the locale object. Like the key name, which is used internally, the user-visible name string indicates the type of data the locale object contains, only it is meant to be displayed to the user.

A names table can include additional names containing text strings that your application can display to your user to describe aspects of the data that the locale object provides, for example, a copyright notice.

When you use the `AddLocaleObject` function (page 2-51) to temporarily add a locale object to the database, you can specify any of the user-displayable names and their identifiers for the locale object.

You use a name-table entry structure to specify a user-displayable name. To specify more than one name-table entry, you pass the function a pointer to an array of name-table entry structures. The `NameTableEntry` data type defines the name-table entry structure.

```
struct NameTableEntry {
    LocaleNameIdentifier nameID;    /* user name identifier*/
    UInt16          reserved;       /* reserved for future use */
```

```
    TextObject      name;           /* text object containing name */
};
typedef struct NameTableEntry NameTableEntry;
```

`nameID` The identifier for the user name specified in the `TextObject` parameter. You use one of the locale object name identifier constants (page 2-13) to specify the name ID.

`reserved` Reserved for future use.

`name` A text object containing the user name text string whose identifier you specified in the `nameID` parameter, the text encoding, and the language and region for the text string. For information on text objects, see "Text Object Manager Reference," to be provided later. For information on text encodings, see "Low-Level Encoding Converter Manager Reference," to be provided later.

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Locale Object Name Identifier Constants

A locale object names tables contains name records for the set of names associated with the locale object. The names table includes a key name for the locale object that the Locale Object Manager uses to catalog the locale object in the database. The key name serves as the primary search key. Two examples of key names are `inputmethod` and `collatetable`.

You use a locale object's key name to specify the primary search key for functions that take a `keyName` parameter. You specify a locale object key name, for example, as part of the search criteria you provide when you call the `LocaleIterateCreate` function (page 2-42).

Most of the names in the names table exist so that you can describe the data contents of a locale object to a user, for example, the user-visible name indicating the type of data and the copyright notice. You can obtain any of these name strings to display to the user by giving the identifier of the name whose data you want.

Each locale object name has associated with it an identifier that serves to identify the type of data the name string contains. The Locale Object Manager defines constants for these identifier names that you can use to specify which of a locale object's names you want to obtain when you call `GetLocaleObjectName`

(page 2-28). When you call `GetLocaleObjectName`, you specify the text encoding used for the name string. Using name identifiers allows a locale object to contain multiple localized name strings.

The Locale Object Manager defines the following six identifier name constants:

```
typedef SInt16  LocaleNameIdentifier;
enum {
    kLocaleObjectKeyNameIdentifier      =   0,  /* identifier for key name */
    kLocaleObjectUserName               =   1,  /* identifier for user-visible name */
    kLocaleObjectCopyrightString        =   2,  /* identifier for copyright value */
    kLocaleObjectManufacturerString     =   3,  /* identifier for manufacturer value */
    kLocaleObjectFunctionDescription    =   4,  /* identifier for function value */
    kLocaleObjectVersionString          =   5   /* identifier for version number
                                                   value */
};
```

Constants descriptions

`kLocaleObjectKeyNameIdentifier`
: Identifier name for the locale object key name. The Locale Object Manager uses this name as a key into the database to find locale objects of this type. You can obtain the locale object key name without using an identifier name by calling `GetLocaleObjectKeyName` (page 2-30). You do not display the key name to the user.

`kLocaleObjectUserName`
: Identifier name for the locale object user-visible name to display to the user.

`kLocaleObjectCopyrightString`
: Identifier name for the copyright value, which is a string you might want to display to the user.

`kLocaleObjectManufacturerString`
: Identifier name for the manufacturer value, which is a string you might want to display to the user.

`kLocaleObjectFunctionDescription`
: Identifier name for the function value that specifies the purpose or type of function provided by the object's data, for example, "U.S. English Configuration Table".

`kLocaleObjectVersionString`
: Identifier name for the version number value that specifies

the version of the locale object's data. For example, a version number value might specify the following string: "Apple Computer Japanese Input Method. Version 1.0".

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Locale Name Identifier for Locale's Default Values

You can obtain default values defined for a locale to display to your user by calling the `GetLocaleInformation` (page 2-35) function and giving the name identifier of the default you are interested in. `GetLocaleInformation` returns a text object from which you can extract the default value text string. To tell `GetLocaleInformation` which of the locale's default values you want, you specify a locale default name identifier. The Locale Object Manager defines the following locale default name identifier constants for a locale's default values.

```
typedef LocaleNameIdentifier LocaleDefaultValue;
enum {
    kLocaleLanguageID               =   0x000A,     /* language id */
    kLocaleLanguageLocalizedName    =   1,          /* localized name of language */
    kLocaleLanguageEnglishName      =   0x000B,     /* English name of language */
    kLocaleAbbreviatedLanguageName  =   0x000C,     /* abbreviated language name */
    kLocaleLanguageNativeName       =   0x000D,     /* native name of language */
    kLocaleCountryCode              =   0x000E,     /* country code */
    kLocaleLocalizedCountryName     =   0x000F,     /* localized name of country */
    kLocaleEnglishCountryName       =   0x1002,     /* English name of country */
    kLocaleAbbreviatedCountryName   =   0x001F,     /* abbreviated country name */
    kLocaleNativeCountryName        =   0x002F,     /* native name of country */
};
```

Field descriptions

`kLocaleLanguageID` A three-character language code from ISO 639.

`kLocaleLanguageLocalizedName`

The fully-localized name for the Locale file. (This is a synonym for `kLocaleObjectUserName`.)

`kLocaleLanguageEnglishName`

The full English name of the locale, from ISO 639. Characters composing the name are restricted to characters in the 7-bit ASCII encoding.

`kLocaleAbbreviatedLanguageName`
: The abbreviated name of the language. (This is a synonym for `kLocaleLanguageID`.)

`kLocaleLanguageNativeName`
: A synonym for `kLocaleObjectUserName` and `kLocaleLanguageLocalizedName`.

`kLocaleCountryCode`
: A two-character code, from ISO 3166.

`kLocaleLocalizedCountryName`
: The fully localized country or territory name.

`kLocaleEnglishCountryName`
: The full English name of the country, from ISO 3166. Characters composing the name are restricted to characters in the 7-bit ASCII encoding.

`kLocaleAbbreviatedCountryName`
: The abbreviated name of the country. For example, U.S is the abbreviated name of the United States.

`kLocaleNativeCountryName`
: A synonym for `kLocaleLocalizedCountryName`.

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Locale Identifier and Constants

A **locale identifier** is a packed value containing packed language and region codes, and a customization code indicating whether the locale is a customized version of a standard locale. Your application can create a locale identifier and use it to obtain a locale reference (page 2-5). You can think of a locale reference as a resolved locale identifier.

You use a locale identifier to indicate the locale for which you want to obtain a reference when you call the `GetLocaleReference` function (page 2-25).

The Locale Object manager returns a locale identifier to you when you use `CreateLocaleIdentifier` to create one. The Locale Object Manager provides a number of functions (page 2-60) that you can use to obtain the locale identifier for locales that reside in the locale database on the current system. Once a locale identifier exists, you can use Locale Object Manager functions to extract the language, region, and customization information that it contains.

A locale identifier is defined by a `LocaleIdentifier` data type.

```
typedef UInt32 LocaleIdentifier; /* locale identifier */
```

The Locale Object Manager defines the following constants that you can use to specify the wildcard locale identifier, the system default locale identifier, or the user default locale identifier.

```
typedef UInt32 LocaleIdentifier;
enum {
    kLocaleIdentifierWildCard       = 0x00000000,
        /* locale identifier wildcard */
    kSystemDefaultLocaleIdentifer   = 0x7FFFFFFF,
        /* locale identifier for system default locale */
    kUserDefaultLocaleIdentifer     = 0x7EEEEEEE
        /* locale identifier for application's default locale */
};
```

Enumerator descriptions

`kLocaleIdentifierWildCard`

Specifies a locale identifier that matches on any language, region, and customization code. If you pass this constant to `GetLocaleReference` (page 2-23), the function returns the locale used for your application's current process.

`kSystemDefaultLocaleIdentifer`

Specifies the system default locale identifier. You can obtain a locale reference to the system locale by passing this constant to the `GetLocaleReference` function.

`kUserDefaultLocaleIdentifer`

Specifies the application's default current locale identifier. You can obtain a locale reference to the user locale by passing this constant to the `GetLocaleReference` function.

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Locale Language Codes and Wildcard

A locale language code is a three-character, lowercase identifier used to indicate a particular written version of a language for the Mac OS 8. The Mac OS 8 recognizes the language codes defined by the International Standards

Organization (ISO) in the "Code For the Representation of Names of Languages, alpha-3 code" dated December 16, 1991 (ISO CD 639/2 draft proposal). For your convenience the constants defined for these codes are included as comments in the `TextCommon.h` file.

If you do not know the language for a particular locale for which you want to create a locale identifier, you can specify the locale language code wildcard when you call the `CreateLocaleIdentifier` function (page 2-61).

The `LocaleLanguageCode` data type defines the language code. The following enumeration defines the locale language code wildcard constant:

```
typedef OSType LocaleLanguageCode;
enum {
    kLocaleLanguageWildCard = 0x00000000 /* locale language wildcard */
};
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for general information.

Locale Region Code and Wildcard

A region code is a two-character, uppercase identifier used to indicate a particular geographical region or territory. The Mac OS 8 recognizes the region codes defined by the International Standards Organization (ISO) in the "Code For the Representation of Names of Languages, alpha-3 code" dated December 16, 1991 (ISO CD 639/2 draft proposal). For your convenience the constants defined for these codes are included as comments in the `TextCommon.h` file.

If you do not know the region for a particular locale for which you want to create a locale identifier, you can specify the locale region code wildcard when you call the `CreateLocaleIdentifier` function (page 2-61).

The `LocaleRegionCode` data type defines the region code. The following enumeration defines the locale region code wildcard constant:

```
typedef UInt16 LocaleRegionCode;
enum {
    kLocaleRegionWildCard   = 0x0000, /*locale region wildcard */
};
```

See “Locale Object Manager Constants and Data Types” (page 2-5) for high-level information.

Locale Customization Code and Wildcard

The Locale Object Manager creates a custom locale based on a locale that exists in the locale database when some aspect of the original locale is changed. The Locale Object Manager assigns a customization code to the locale identifier for the new version of the locale. For example, if a French-Canadian locale is modified in some way—suppose any of the default values for the locale, such as a number separator, have been changed—the Locale Object Manager would create a new custom version of the locale and assign it a customization code.

The Locale Object Manager sets a customization code internally to indicate that the locale is a customized version of a standard system locale. Because this value is set internally, you should always specify the `kLocaleCustomizationWildCard` constant for the `CreateLocaleIdentifier` function (page 2-61) when you call the function to create a locale identifier.

The `LocaleCustomizationCode` data type defines the customization code. The following enumeration defines the locale customization code wildcard constant:

```
typedef UInt16 LocaleCustomizationCode;
enum {
    kLocaleCustomizationWildCard = 0x0000
        /* locale customization wildcard */
};
```

See “Locale Object Manager Constants and Data Types” (page 2-5) for general information.

Locale Object Tag Index

A locale object can contain additional data related to its primary data. For example, this associated data might contain an icon for the primary data or it might contain additional user-displayable names beyond the set allowed in a names table (page 2-13) for a locale object. Each collection of associated data for a locale object is identified by a four-character tag that is stored along with the data.

A locale object may have associated with it multiple collections of associated data. To obtain data associated with a locale object's primary data, you identify the type of associated data by specifying the data's tag. If you don't know the tag name for a collection of data, you can use the `GetIndexedAssociatedData` function (page 2-57) to retrieve additional data associated with a locale object by specifying the zero-based index number of the tag. You can use `CountLocaleObjectAssociatedDataTags` (page 2-56) to get the number of collections of data using the count to increment through the tag indices.

The Locale Object Manager defines the following data type that you use to specify a tag index:

```
typedef UInt32  LocaleObjectTagIdentifier;
    /* locale associated data type tag */
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Associated-Data Tag

A locale object can contain additional data associated with its primary data. For example, this data might specify an icon for the primary data or it might contain additional user-displayable names beyond the set allowed in a names table (page 2-13) for a locale object. Each collection of associated data for a locale object is identified by a four-character tag that is stored along with the data.

When you want to obtain data associated with the primary data of a locale object, you pass the `GetLocaleObjectAssociatedData` function the associated data's tag. The `GetIndexedAssociatedData` function (page 2-57) returns the data tag and data you identify by index.

The Locale Object Manager defines the following data type for specifying an associated-data tag:

```
typedef OSType LocaleDataTag; /* associated data tage */
```

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Locale Object Memory Context

A locale object can reside in system-wide memory (global memory) or it can reside in your application's per-process memory heap (local memory). To determine where in memory a locale object resides, you call the `GetLocaleObjectMemoryContext` function (page 2-72), which returns one of the following constants to identify the object's memory location.

```
typedef UInt16  LocaleObjectContext;
enum {
    kLocaleObjectIsGlobal   =   0,
        /* locale object resides in system-wide memory */
    kLocaleObjectIsLocal    =   1
        /* locale object resides in application's per-process heap */
};
```

Enumerator descriptions

`kLocaleObjectIsGlobal`

Identifies the locale object as residing in system-wide memory.

`kLocaleObjectIsLocal`

Identifies the locale object as residing in your application's per-process memory heap.

See "Locale Object Manager Constants and Data Types" (page 2-5) for high-level information.

Locale Object Manager Functions

Obtaining and Setting Locale References

You use a locale reference to specify the locale you are interested in when you call the Locale Object Manager functions to access and act on the data contained in locales and to change the default locale to be used within your application's CFM context. You can think of a locale reference as a resolved locale identifier. The Locale Object Manager includes these functions that you can use to obtain locale references:

- `GetCurrentProcessLocaleRef` returns the reference to the default locale for your application's current process.
- `GetLocaleReference` returns a reference to the locale whose primary language and region match those you specify in the locale identifier.
- `GetSystemDefaultLocaleRef` returns the locale reference for the current system default locale.

GetCurrentProcessLocaleRef

Returns the locale reference to the default locale for the current process.

```
LocaleRef GetCurrentProcessLocaleRef (void);
```

function result The locale reference (page 2-5) for the default locale of the current process.

DISCUSSION

The `GetCurrentProcessLocaleRef` function returns the reference to the locale used for your application's current process, that is, the locale used for the current Code Fragment Manager (CFM) context. You can think of this as the default locale for your application. At system startup, the Locale Object Manager establishes the default locale for the application based on the default system locale. You can change the locale for the current process using the `SetCurrentProcessLocale` function (page 2-26). After you a set new locale for the current process, that locale becomes the default locale; if you call `GetCurrentProcessLocaleRef` subsequently, the function returns a reference to the newly set locale.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleReference

Returns a reference to the locale whose primary language and region match those you specify in the locale identifier.

```
OSStatus GetLocaleReference (LocaleIdentifier identifier,
                    LocaleRef *locale);
```

`identifier` A locale identifier specifying the primary language and region for the locale whose reference you want to obtain. To obtain a locale identifier for a locale resident in the database, use one of the functions (page 2-60) provided for this purpose. To obtain a reference to the system default locale, specify `kSystemDefaultLocaleIdentifier` (page 2-16). To obtain a reference to your application's default current locale, specify `kUserDefaultLocaleIdentifer` (page 2-16). To obtain a reference to the first locale in the database, specify `kLocaleIdentifierWildcard` (page 2-16).

`locale` A pointer to a locale reference. On output, this pointer refers to the locale reference (page 2-5) for the locale you identified.

function result A result code. If the locale database does not include a locale for the locale identifier you specify, the function returns a `localeNotFoundErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

The `GetLocaleReference` function returns a locale reference that refers to a specific locale within the locale database. To identify the primary language and region characterizing the locale for which you want a locale reference, you supply a locale identifier. To obtain a locale identifier, you can use any of the Locale Object Manager functions (page 2-60) that return locale identifiers for locales resident in the database.

Note
If you use the `CreateLocaleIdentifier` function to create a locale identifier for a locale that is not resident in the database, you should not call `GetLocaleReference` using that locale identifier. Because the locale is not resident, there is no locale reference for it. ◆

Many of the Locale Object Manager functions require that you specify a locale reference to identify a locale that you want to act on in some way. You use a locale reference to position the start of a locale database search. For example, you pass a locale reference to `SearchOneLocaleObject` (page 2-40) to identify the locale where you want to begin a search for the first locale object that matches your criteria. You pass a locale reference to `LocaleIteratorCreate` (page 2-42) to identify the locale where you want to begin an iterative search throughout the locales of the database for data and information belonging to one or more locale objects. You pass a locale reference to `SetLocaleIterator` (page 2-45) to change the starting position of an iterative search.

You identify the locale to which you want to add a locale object for your use within the current process by specifying its locale reference when you call the `AddLocaleObject` function (page 2-51).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetSystemDefaultLocaleRef

Returns the locale reference for the default system locale.

```
LocaleRef GetSystemDefaultLocaleRef (void);
```

function result The locale reference (page 2-5) for the default system locale.

DISCUSSION

At system startup, the Locale Object Manager establishes the default system locale based on the language and region for which the system is localized. You can use `GetSystemDefaultLocaleRef` to obtain a reference to that locale.

You might want to use this function, for example, to obtain a reference to the system locale if your application customized the system locale using `AddLocaleObject` (page 2-51) for the duration of your process, but within that process you want to revert to using the default system locale.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

The default system locale becomes the default locale for your application. However, you can change the locale used for your application by calling the `SetCurrentProcessLocale` function (page 2-26).

Setting the Locale for the Current Process

SetCurrentProcessLocale

Sets the locale to be used for the current process, that is, the current CFM (Code Fragment Manager) context, and returns a reference to it.

```
OSStatus SetCurrentProcessLocale (LocaleIdentifier localeID,
                    LocaleRef *locale);
```

`localeID` A locale identifier (page 2-16) for the locale to be used for the current process.

`locale` A pointer to a locale reference (page 2-5). On output, the pointer refers to the locale reference for the locale to be used for the current process.

function result A result code. If the locale identifier you specify in the `localeID` parameter is invalid, the function returns an `localeNotFoundErr` result code and the locale for the current process is not changed. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

Your application can use `SetCurrentProcessLocale` to change the locale for the current process, that is, the current Code Fragment Manager (CFM) context. You might want to do this, for example, if your application is localized for one system, but it is running on a system localized for a different country or region. At system startup, the Locale Object Manager establishes the default locale for the application based on the default system locale. At system startup, the Locale Object Manager establishes the default locale for the application based on the default system locale.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use any of the functions that the Locale Object Manager provides to obtain a locale identifier (page 2-60) to supply one to `SetCurrentProcessLocale`.

Obtaining the Number of Locales in the Database

CountInstalledLocales

Returns the number of locales installed in the locale database on the system.

```
ItemCount CountInstalledLocales (void);
```

function result The number of locales in the database.

DISCUSSION

The function counts the locales in the database and returns this number. You might want to use this number, for example, to determine how much memory to reserve for information such as locale references returned by `GetLocaleReference` (page 2-23) if your application calls `GetLocaleReference` from within a loop to move through the locales of the database and obtain a reference for each one.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Obtaining a Locale Object's Name, Attributes, Data, and Locale

You can use the following functions to obtain information about a locale object:

- `GetLocaleObjectName` finds a localized version of a locale object name and returns it as text object.
- `GetLocaleObjectKeyName` returns the key name for the locale object whose locale object reference you specify.
- `GetLocaleObjectAttributes` returns all attributes of the specified object.
- `GetLocaleObjectData` returns the data stored in the data portion of the specified locale object.
- `GetLocaleObjectLocale` identifies the locale to which the specified locale object belongs, returning the locale reference and the locale identifier for the locale.

GetLocaleObjectName

Returns a text object containing an object name.

```
OSStatus GetLocaleObjectName (LocaleObjectRef objectRef,
                    LocaleNameIdentifier nameID,
                    TextEncoding encoding,
                    LocaleIdentifier languageRegion,
                    ByteCount *nameSize,
                    TextObject name);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose name string you want to obtain as a text object.

`nameID` A locale name identifier (page 2-13) that specifies one of the locale object names. To identify which name you want, specify one of the name identifier constants defined by the Locale Object Manager.

`encoding` The text encoding of the name identified by the `nameID` parameter. A text encoding is a value that describes the text encoding scheme character set and its packaging format used to represent the text. The Locale Object Manager stores the text encoding you provide along with the name string in the text object. For information on text encodings, see "Low-Level Encoding Converter Manager Reference," to be provided later.

`languageRegion` A locale identifier (page 2-16) that specifies the language and region for which the name that you want is localized. The Locale Object Manager stores the information you provide along with the name string in the text object.

`nameSize` On input, a pointer to a value of type `ByteCount`. If you want to use a persistent text object, pass in a pointer. If you use an ephemeral text object—for example, you pass the result from a call to `NewTextObject` as the value of `name`—specify `NULL` for this parameter on input. On output, `nameSize` returns the size in bytes of the persistent text object.

`name` A text object (see "Text Object Manager reference" to be provided later.) On input, an ephemeral text object or `NULL` if you want the function to return the name in a persistent text object. On output, the text object contains the name string corresponding to the name identification given in the `nameID` parameter, the text encoding, and the language and region for the text string.

DISCUSSION

A locale object contains a name table that can include up to six name records. The name table includes records for the locale object's key name and user-displayable names, such as the locale object user name and copyright information. A name record contains a name ID field that tells the type of

information its name string field contains. The Locale Object Manager defines constants to identify the names of the name table. You give a name identifier constant for the `nameID` field to specify which of the name strings belonging to the locale object you want `GetLocaleObjectName` to return as a text object.

For example, to obtain a text object to display to your user containing the locale object user name, you would specify `kLocaleObjectUserName` as the name identifier when you call `GetLocaleObjectName`. A text object is a private data type that you use to pass data to the Mac OS 8 system components and to store text displayed as part of your user interface.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleObjectKeyName

Returns the key name for the locale object whose locale object reference you specify.

```
OSStatus GetLocaleObjectKeyName (LocaleObjectRef objectRef,
                    Str255 keyName);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose key name you want to obtain.

`keyName` On output, a text string giving the key name (page 2-13) for the specified locale object.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can use the locale object key name that `GetLocaleObjectKeyName` returns as part of the criteria to search the locale database for matching locale objects. Key names usually describe the type of data the locale object contains, for example, *collatetable*.

You provide a locale object key name to `LocaleIteratorCreate` (page 2-42) to be used in an iterative search for locale objects. You provide a locale object key name to `SetLocaleIterator` (page 2-45) to change that portion of the existing search criteria. To search for the first matching locale object found, you provide a locale object key name to the `SearchOneLocaleObject` function (page 2-40).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleObjectAttributes

Returns all attributes of the specified object.

```
OSStatus GetLocaleObjectAttributes (LocaleObjectRef objectRef,
                    const NameValuePair **attributes,
                    ItemCount *countPairs);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose attributes you want to obtain.

`attributes` A pointer to an attribute name-value pair structure (page 2-8). On output, the pointer refers to the table in memory containing the full set of attribute name-value pairs associated with the specified locale object. This table is read only.

`countPairs`	A pointer to a value of type `ItemCount`. On output, the value pointed to specifies the number of attribute name-value pairs contained in the table referred to by the `attributes` parameter.
function result	A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

Once you identify the locale object whose attributes you want to obtain, you provide its locale object reference to the `GetLocaleObjectAttributes` function. If the function completes successfully, `GetLocaleObjectAttributes` returns a pointer to the set of attributes associated with the specified locale object and a pointer to the number of attribute pairs contained in that set. The attribute name-value pairs set pointed to in memory is read-only; you cannot write to it. However, you can use the number of pairs returned as a counter to increment through the table in memory. You can read the attributes data or copy it as you increment through the table.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleObjectData

Returns a pointer to the data stored in the data portion of the specified locale object.

```
OSStatus GetLocaleObjectData (LocaleObjectRef objectRef,
                    const void **localeObjectData,
                    ByteCount *dataSize);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose data you want to obtain.

`localeObjectData`
A pointer to data. On output, the pointer refers to the data belonging to the locale object whose reference you specify. The data pointed to is read only.

`dataSize` A pointer to a value of type `ByteCount`. On output, the value pointed to gives the size in bytes of the data referred to by the `localeObjectData` parameter.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

Once you identify the locale object whose data you want to obtain, you pass its locale object reference to the `GetLocaleObjectData` function to obtain a pointer to the object's data. A locale object's data might consist of a sorting table or a number format table, for example.

The data pointed to is read only. You can copy the data and write to the copy of it, but you must not alter the original data in memory. This rule applies unless the locale object whose reference you provide is a locale object that your application added to the database for its use within the current process; in this case, you can modify the original data you provided.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleObjectLocale

Identifies the locale to which the specified locale object belongs, returning the locale reference and the locale identifier for the locale.

```
OSStatus GetLocaleObjectLocale (
                    LocaleObjectRef objectRef,
                    LocaleRef *locale,
                    LocaleIdentifier *localeID);
```

`objectRef` The locale object reference (page 2-8) to the locale object whose locale you want to know.

`locale` A pointer to a locale reference (page 2-5). On output, the pointer refers to the locale reference of the locale to which the specified locale object belongs.

`localeID` A pointer to a locale identifier (page 2-16). On output, the pointer refers to the locale identifier that specifies the language and region for the locale to which the locale object belongs.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

When you perform an iterative search of the locale database and you obtain a locale object that satisfies your matching criteria, you don't know to which locale the object belongs because the search is not constrained to one locale.

You can use `GetLocaleObjectLocale` to determine the locale to which the object belongs and to obtain the locale identifier from which you can determine the primary language and region of the locale.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To determine the primary language and region of the locale, use the `GetLocaleLanguage` (page 2-68) and `GetLocaleRegion` (page 2-69) functions.

Obtaining a Locale's Default Values

GetLocaleInformation

Returns a text object containing the default value indicated by the name identifier you specify for the locale whose reference you specify.

```
OSStatus GetLocaleInformation (LocaleRef locale,
                    LocaleDefaultValue infoID,
                    TextObject *infoText);
```

`locale` A locale reference identifying the locale for which you want the name of the default value.

`infoID`	The identifier of the locale name whose default value you want returned in the text object. To specify the default value name identifier, use one of the locale name default value (page 2-15) constants.
`infoText`	A pointer to a text object. On output, the text object contains the default value name string corresponding to the name identifier given in the `infoID` parameter. For information on text objects, see "Text Object Manager Reference," to be provided later.
function result	A result code. See "Locale Object Manager Result Codes" (page 2-72).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Getting and Setting Default Behavior for a Locale

When the Locale Object Manager initially builds the locale database, it sets default behaviors for each locale based on information provided by a localizer in a Locale file. For a given type of data identified by a key name, you can obtain or set the default for a locale. You use the `GetDefaultLocaleObject` function to obtain a reference to the locale object containing the locale's default data for a specific behavior. You use the `SetDefaultLocaleObject` function to set a locale's default data for a specific behavior.

GetDefaultLocaleObject

Returns a reference to the locale object containing the locale's default data for the behavior specified by the key name you provide.

```
OSStatus GetDefaultLocaleObject (
                    LocaleRef locale,
                    ConstStr255Param keyName,
                    LocaleObjectRef *objectRef);
```

`locale` The locale reference (page 2-5) to the locale for which you want the default data.

`keyName` A text string giving the key name (page 2-13) of the type of data for which you want the default locale object.

`objectRef` A pointer to a locale object reference (page 2-8). On output, the locale object reference to the default locale object for the type of function specified by `keyName`.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can use this utility to obtain the default data that determines the behavior of the locale for any type of text-handling operation. You specify the locale object's key name to identify the default behavior you want to know about. Default behaviors for a given locale are based on information provided in a Locale file created by a localizer and read by the Locale Object Manager when it initially builds the database.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SetDefaultLocaleObject

Sets the specified locale's default behavior within the current process for the kind of operation identified by the given key name contained in the given locale object reference.

```
OSStatus SetDefaultLocaleObject (
                    LocaleRef locale,
                    LocaleObjectRef objectRef);
```

`locale` The locale reference (page 2-5) to the locale for which you want to set the default.

`objectRef` A locale object reference (page 2-8) to the locale object containing the data to use for the default behavior of the operation specified by the locale object's key name.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Searching for the First Matching Object of a Locale and Searching Iteratively

You can search the locale database for one or more locale objects whose data you want and obtain a reference to the locale object, a pointer to its data, or both. You can use the pointer to access the locale object's data directly after calling the search function, or you can preserve the locale object reference and pointer use them later, delaying the retrieval of the data. The Locale Object Manager provides these functions for searching the database:

- `SearchOneLocaleObject` finds the first locale object that matches your search criteria.
- `LocaleIteratorCreate` creates an iterator containing the search criteria for an iterative search.
- `LocaleIteratorCreate` iterates through the database to find as many matching locale objects as you want.
- `SetLocaleIterator`modifies the content of an existing locale iterator reference and resets its starting position.
- `LocaleIteratorDispose` releases the memory for the iterator when you no longer need it.

SearchOneLocaleObject

Using the locale object's key name and attributes you provide, searches the locale database for the first matching locale object and returns a reference to it along with a pointer to that locale object's data.

```
OSStatus SearchOneLocaleObject(LocaleRef locale,
                    ConstStr255Param keyName,
                    UInt16 countAttributes,
                    const NameValuePair *attributes,
                    const void **localeObjectData,
                    ByteCount *dataSize,
                    const LocaleObjectRef *objectRef);
```

`locale` A locale reference (page 2-5) to the locale where the search is to begin. The Locale Object Manager starts the search here and traverses the database until it finds the first matching locale object.

`keyName` The key name (page 2-13) for the specified locale object.

`countAttributes` The number of attribute name-value pairs that you provide in the `pairs` parameter.

`attributes` A pointer to an array of attribute name-value pair structures (page 2-8). On input, this array provides the attributes that the returned locale object must possess to satisfy the search criteria.

`localeObjectData` A pointer to data. On output, the pointer refers to the data belonging to the locale object that satisfied the matching criteria. The data pointed to is read only. (The `objectRef` parameter returns a reference to the locale object whose data is pointed to by this parameter.) If you don't want the function to return a pointer to the data, pass in `NULL` for this parameter.

`dataSize` A pointer to a value of type `ByteCount`. On output, the value pointed to contains the size in bytes of the data returned in the `localeObjectData` parameter. If you don't want the function to return this value, pass in `NULL` for this parameter.

`objectRef` A pointer to a locale object reference (page 2-8). On output, the pointer refers to the locale object reference for the first locale object found in the locale database that meets the matching criteria. This is the locale object whose data is provided in the `localeObjectData` parameter. If you don't want the function to return a locale object reference, pass in `NULL` for this parameter.

function result A result code. If the locale database does not include a locale object that meets the matching criteria, the function returns an `localeObjectNotFoundErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

The `SearchOneLocaleObject` function lets you search the locale database for the first locale object that matches the key name and attribute name-value pairs criteria that you provide. You can position the start of the search anywhere in the database by giving the locale reference to the locale where you want the Locale Object Manager to begin. `SearchOneLocaleObject` returns only one matching locale object, unlike `LocaleIterate` (page 2-47), which you can use to search the entire database for all matching locale objects. However, you do not need to create a locale iterator reference (page 2-6) for use with `SearchOneLocaleObject` as you must for use with `LocaleIterate`.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain a reference to the locale where you want to position the start of the search, you use `GetLocaleReference` (page 2-23), passing it a locale identifier (page 2-6) to denote the locale.

LocaleIteratorCreate

Creates and returns a locale iterator reference that you can use to iteratively search the locale database for objects having characteristics you specify as matching criteria.

```
OSStatus  LocaleIteratorCreate (LocaleRef locale,
                    ConstStr255Param keyName,
                    ItemCount countAttributes,
                    const NameValuePair *attributes,
                    LocaleIteratorReference *localeIteratorRef);
```

`locale` A locale reference (page 2-5) to the locale with which the Locale Object Manager is to begin the search. This parameter sets the locale in the locale iterator reference that designates where `LocaleIterate` (page 2-47) will position the start of the search. To begin the search with the default locale of the current process, set this parameter to `NULL`.

`keyName` A locale object key name (page 2-13) that you provide to specify part of the matching criteria used in the search. For a locale object to satisfy this part of the matching criteria, its key name, specified in its name table, must match the key name you supply. Together the locale object key name and the list of attribute name-value pairs you give in the `pairs` parameter constitute the matching criteria. To base the search on only the attribute name-value pairs, specify `NULL` for this parameter.

`countPairs` The number of attribute name-value pairs given in the `pairs` parameter. If you do not provide attribute name-value pairs as part of the search criteria, specify 0 for this parameter.

`pairs` A pointer to an array of attribute name-value pair structures (page 2-8). On input, this array provides the attributes that the returned locale object must possess to satisfy the search criteria.

Together the attribute name-value pairs and the locale object key name constitute the matching criteria. To base the search on only the locale object key name you give in the `keyName` parameter, specify `NULL` for this parameter.

`localeIteratorRef`

A pointer to a locale iterator reference (page 2-6). On output, this pointer refers to the locale iterator reference created with the search criteria and starting position you specified.

function result A result code. If the locale reference you specify in the `locale` parameter is invalid, the function returns a `localeBadReferenceErr` result code. If there is not enough memory available to create the locale iterator reference, the function returns a `memFullErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

Your application can gain access to different types of data stored in the locale database by using the Locale Object Manager to search the database. To search iteratively throughout the database, you use a private data structure called a locale iterator reference.

To search through the locale database, you pass a locale iterator reference to the `LocaleIterate` (page 2-47) function. In response, the Locale Object Manager finds locale objects in the locale database that match your description. When `LocaleIterate` encounters a locale object that matches your description, it returns a locale object reference (page 2-8) and the data for the matching object to your application. You can continue to call `LocaleIterate` from within a loop to find all matching locale objects using the same locale iterator reference until you find what you are looking for. The locale iterator reference used for the search tracks the progress through the locale database maintaining the next position at which to continue the search. When there are no more matching locale objects in the database, `LocaleIterate` returns an `eObjectNotFound` result code that you can test against.

To create a locale iterator reference, you use `LocaleIteratorCreate`, passing it the matching criteria that describes what it is you are looking for. For this matching criteria, you typically provide a locale object key name, one or more attributes, or both. However, to look at all objects in the locale database, you

can create a locale iterator reference that specifies only the starting position of the search, but no matching criteria. In this case, you would specify `NULL` for both the `keyName` and `pairs` parameters and 0 for the `countPairs` parameter.

A locale object key name describes the type of data you are interested in. For example, you might want to find all input methods, in which case you would specify `inputmethods` as the `keyName` parameter. You might want to refine the search further by specifying one or more attributes that a qualifying locale object must possess, such as an attribute designating the language that the input method supports. In this case, you would provide an attribute name-value pair structure specifying `kLanguageName` as the name ID, the size in bytes of the language code, and the language code for the specific language you are interested in.

As part of the locale iterator reference, you can specify where in the database you want the search to begin by giving a reference to the locale that serves as the starting point. Positioning the start of the search at a specific locale allows you greater efficiency if you do not want to search the entire database. For example, the Locale Object Manager may encounter and return the specific locale object you are looking for before it has worked its way through half the locales composing the database. However, it is important to understand that the locale whose reference you provide serves only as the starting point and does not limit the search to that locale. The search proceeds from that locale forward or backward throughout the locales of the database depending on the direction you specify when you call `LocaleIterate`.

A locale iterator reference remains available for your use until you dispose of it by calling `LocaleIteratorDispose` (page 2-50).

You can use the same locale iterator data structure for multiple, distinct searches. For each new search, you can change the search matching criteria—the locale object name and its attributes—of an existing locale iterator. However, for each new search, you must remember to reset the starting position using the `SetLocaleIterator` (page 2-45) function, otherwise the search will fail.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SetLocaleIterator

Modifies the content of an existing locale iterator reference, changing or resetting either the search's starting position only, or its starting position and its matching criteria.

```
OSStatus SetLocaleIterator (LocaleRef locale,
                    ConstStr255Param keyName,
                    ItemCount countAttributes,
                    const NameValuePair *attributes,
                    LocaleIteratorReference *localeIteratorRef);
```

`locale` A locale reference (page 2-5) to the locale with which the iterator is to begin the search. This parameter resets the starting position for the search in the locale iterator reference specified by the `localeIteratorRef` parameter. You can reset the locale reference to restart a search. You must always reset the starting position when you use an existing locale iterator reference for a new search. You can reset the iterative search to its original starting position by specifying `NULL` for this parameter instead of specifying a new locale reference.

`keyName` A locale object key name (page 2-13) that you provide to change the name string currently set in the specified locale iterator. A locale object key name serves as part of the matching criteria used in the search. For a locale object to satisfy this part of the matching criteria, it must include this name among the names belonging to it. To reuse the existing locale object key name set

for the locale iterator reference, specify -1 for this parameter. Together the locale object key name and the list of attribute name-value pairs contained in the locale iterator reference constitute the matching criteria.

`countAttributes`
The number of attribute name-value pairs given in the `pairs` parameter.

`attributes` A pointer to an array of attribute name-value pair structures (page 2-8). On input, this pointer refers to an array containing the attributes that you provide to replace the existing name-value pairs in the specified locale iterator. To reuse the existing attribute name-value pairs in the locale iterator reference, specify -1 for this parameter.

`localeIteratorRef`
A pointer to a locale iterator reference (page 2-6). On input, the pointer refers to an existing locale iterator reference whose content you want to modify.

function result A result code. If the locale reference you specify in the `locale` parameter is invalid, the function returns a `localeBadReferenceErr` result code. The function returns a `memFullErr` result code if there is insufficient memory for the Locale Object Manager to modify the locale iterator reference.

DISCUSSION

After you have created a locale iterator reference using `LocaleIterateCreate` (page 2-42), you can use `SetLocaleIterator` to modify its content. To reuse an existing locale iterator reference, you must always reset the starting position specified by the `locale` parameter, otherwise the search will fail. The Locale Object Manager traverses and tracks its progress through the entire locale database once using a locale iterator reference. Resetting the starting position informs the Locale Object Manager that you want to use the existing locale iterator reference for a new search so that it can clear any tracking information used for the last search. If you want to position the search at the locale currently specified by locale iterator reference, pass `NULL` as the `locale` parameter. To reuse the existing locale object name or the existing set of attributes, specify -1 for these parameters. You can use all the current

information in the locale iterator reference to search the database again by specifying `NULL` for `locale`, -1 for `keyName`, and -1 for `pairs`.

You can change the locale object key name or the attribute name-value pairs, or both parts of the search criteria. If you supply new attribute name-value pairs, you replace all of the existing ones formerly specified; you cannot partially modify a set of attributes other than by providing the same attribute along with the new ones you supply in your array.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

LocaleIterate

Searches the locale database iteratively for locale objects that match the search criteria specified by the locale iterator reference you provide, and returns a matching locale object's reference and a pointer to its data.

```
OSStatus  LocaleIterate (LocaleIteratorReference localeIteratorRef,
                         LocaleIterateOp op,
                         const void **dataPtr,
                         ByteCount *dataSize,
                         const LocaleObjectRef *objectRef);
```

`localeIteratorRef`
A locale iterator reference (page 2-6) containing the search criteria and starting position.

`op`	The direction in which you want the search to proceed, either forward or backward. You can use the locale database search direction constants (page 2-7) for this parameter.
`dataPtr`	A pointer to data. On output, this pointer refers to the read-only data belonging to the matching locale object if `LocaleIterate` finds a locale object that satisfies the search criteria. You can copy this data, but you cannot modify it in the memory location pointed to by this parameter. If no matching locale object exists in the database, this parameter contains an invalid value and the function returns an `eObjectNotFound` result code.
`dataSize`	A pointer to a value of type `ByteCount`. On output, the value pointed to specifies the size in bytes of the locale object's data pointed to by the `dataPtr` parameter.
`objectRef`	A pointer to a locale object reference. On output, the locale object reference for the locale object that matched the search criteria.
function result	A result code. When there are no more locale objects in the locale database that match the description you provide in the locale iterator reference, or if the first time you call `LocaleIterate` it searches the entire database without finding a matching locale object, `LocaleIterate` returns an `localeNotFoundErr` result code. If the locale iterator reference you specify is `NULL`, the function returns a `paramErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

Your application can use an iterative search to gain access to the data of all locale objects in the database that have in common a set of characteristics. To simplify this process and allow you to widen or narrow the search criteria more easily, the Locale Object Manager allows you to create a locale iterator reference that holds both the characteristics of the locale objects whose data you are interested in and the position in the database where the search is to begin. You use `LocaleIterateCreate` (page 2-42) to create a locale iterator reference that you pass to the `LocaleIterate` function when you call `LocaleIterate` from within a loop to search iteratively through the database for matching locale objects.

The `LocaleIterate` function begins at the locale whose reference you specified when you created the locale iterator reference, but it proceeds from there to search through all locales of the entire locale database for locale object matches. `LocaleIterate` returns a reference to the locale object and a pointer to its data each time the function finds a match until it completes its search of the entire database or until you stop the search because you have found the locale object you were searching for. The locale object data pointed to by the `dataPtr` parameter is read only; your application can copy its contents, but you cannot write to it in memory.

The locale iterator used for the search tracks the progress through the locale database maintaining the next position at which to continue the search. You can specify the direction of the search when you call the function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use the locale object reference returned by `LocaleIterate` to specify the locale object whose attributes you want to obtain when you call `GetLocaleObjectAttributes` (page 2-31).

LocaleIteratorDispose

Disposes of a locale iterator reference created by the `LocaleIteratorCreate` function.

```
OSStatus LocaleIteratorDispose (
                    LocaleIteratorReference localeIteratorRef);
```

`localeIteratorRef`
The locale iterator reference to be disposed of.

function result A result code. If you specified an invalid locale iterator reference, `LocaleIteratorDispose` returns a `paramErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can reuse existing locale iterator references for multiple, distinct searches, but when you are entirely finished with one, you must dispose of the memory used for the locale iterator reference by calling this function.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Adding Locale Objects To and Removing Them From the Locale Database

You can add data to the locale database and remove it from the database from within your application's current process. You use the `AddLocaleObject`

function to add a locale object and the `RemoveLocaleObject` function to remove any locale object you added.

AddLocaleObject

Adds a locale object to the specified locale for the duration of the current process, and returns a reference to the object.

```
OSStatus AddLocaleObject(LocaleRef locale,
                    void *localeObjectData,
                    ByteCount objectSize,
                    ConstStr255Param keyName,
                    ItemCount countUserNames,
                    const NameTableEntry *userName
                    ItemCount countAttributes,
                    const NameValuePair *attributes,
                    LocaleObjectRef *objectRef);
```

`locale` A locale reference to the locale (page 2-5) to add the specified locale object to.

`localeObjectData`
A pointer to data. On input, the pointer refers to the data for the new locale object to be added to the database. You can provide any type of data for the locale object; the `localeObjectData` data type is a void pointer, so it doesn't restrict the type of data you can supply.

`objectSize` The size in bytes of the locale object's data given in the `localeObjectData` parameter.

`keyName` A locale object key name for the new locale object.

`countUserNames`
The number of user names you provide in the `userNames` parameter.

`userNames` A pointer to a name-table entry structure (page 2-12). On input, the pointer refers to the first name-table entry in the array of structures containing the user names and their identifiers for the data you are adding. You must specify at least a locale object user name (`kLocaleObjectUserName`).

`countAttributes` The number of attribute name-value pairs you provide in the `attributes` parameter. If you are not providing any attributes, specify 0 for this parameter.

`attributes` A pointer to an attribute name-value pair structure (page 2-8). On input, this pointer refers to the first attribute name-value pair in the array of structures containing the attributes for the locale object to be added. If you are not providing any attributes, pass in `NULL` for this parameter.

`objectRef` A pointer to a locale object reference (page 2-8). On output, a locale object reference to the new locale object temporarily added to the database.

function result A result code. If there is not enough memory available for the function to create the new locale object and add it to the database, the function returns a `memFullErr` result code. If a locale object having the same locale object key name and attributes you specify for the new one already exists in the database, the function returns a `localeObjectNameAttributeConflictErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can add a locale object to a specific locale to be used within the current process, that is, the current CFM (Code Fragment Manager) context. The locale object is temporarily incorporated in the database.

You should always use the `RemoveLocaleObject` function (page 2-53) to explicitly remove the locale object from the database before the current process terminates. Then, you should dispose of the memory you allocated for the parts of that object—its data, key name, user names, and attributes.

If you do not explicitly remove the locale object from the database before the process terminates, the Locale Object Manager removes it when this occurs. In

this case, you are still responsible for releasing the memory allocated for the object parts.

Adding a locale object to the database using `AddLocaleObject` is the way to temporarily replace an existing locale object—for example, to install a custom sorting table to override the default one for the locale for the duration of the current process.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

RemoveLocaleObject

Removes the specified locale object from the locale database.

```
OSStatus RemoveLocaleObject (LocaleObjectRef objectRef);
```

`objectRef` A locale object reference (page 2-8) to the locale object to be removed from the database.

function result A result code. If you specify an invalid locale object reference, the function returns a `localeObjectInvalidReferenceErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can use this function to remove locale objects that you added to the locale database, but you cannot remove locale objects added to the locale database when it was built at system startup or locale objects added to it by other clients

of the database. Removing a locale object from the database does not release the memory you allocated for the object's parts when you added the locale object using `AddLocaleObject` (page 2-51). It is your responsibility to dispose of the data and release the memory for it.

You should always call this function to remove a locale object that you added within the context of the current process before that process terminates.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Getting Data Associated With a Locale Object

You can obtain additional data associated with a locale object's primary data. If you know the tag that identifies the collection of data you want, you can use the `GetLocaleObjectAssociatedData` function to obtain it. If you don't know the tag, you can obtain the tags for a locale object's associated data by first determining the total number of tags, then using the number as a count to increment through the tags. To identify the total number of tags, you use the `CountLocaleObjectAssociatedDataTags` function. To obtain a tag based on its index, you use the `GetIndexedAssociatedData`.

You can use the `GetLocaleObjectFSObjectRef` function to obtain the file specification object reference for the file that originally contained a given locale object.

GetLocaleObjectAssociatedData

Returns a pointer to the tag-identified data associated with the locale object whose reference you specify.

```
OSStatus GetLocaleObjectAssociatedData (LocaleObjectRef objectRef,
                    LocaleDataTag tag,
                    const void **associatedDataPtr,
                    ByteCount *size);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose data you want to obtain.

`tag` A data tag (page 2-20), consisting of four characters enclosed in single quotation marks, identifying the associated data you want.

`associatedDataPtr`
A pointer to data. On output, this pointer refers to the data whose tag you specified.

`size` A pointer to a value of type `ByteCount`. On output, this value specifies the size of the associated data.

function result A result code. If the locale object reference that you specify in the `objectRef` parameter is invalid, the function returns a `localeObjectInvalidReferenceErr` result code. For other possible returned result codes, see "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

After you search for a locale object and obtain a reference to it using either `SearchOneLocaleObject` (page 2-40) or `LocaleIterate` (page 2-47), you can call `GetLocaleObjectAssociatedData` to obtain any collection of data associated with the locale object's primary data.

A locale object may have associated with it multiple collections of associated data. To identify the type of associated data you want to obtain, you specify the data's tag.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

CountLocaleObjectAssociatedDataTags

Returns the number of associated-data tags that exist for the specified locale object.

```
ItemCount CountLocaleObjectAssociatedDataTags (
                    LocaleObjectRef objectRef);
```

`objectRef` A locale object reference (page 2-8) to the locale object for which you want the number of tags.

function result The number of associated-data tags belonging to the specified locale object. If there is no data associated with the locale object, this function returns a value of 0.

DISCUSSION

A locale object can have additional data associated with it that is identified by a 4-character tag. If you want to obtain any collection of associated data included in a locale object for use with its primary data, you need to know the associated data's tag. You can obtain the tags for associated data by first calling `CountLocaleObjectAssociatedDataTags` to get the total number of tags. You can then use this number as a count to increment through the tags referring to them by index based on this count. You use this function in conjunction with `GetIndexedAssociatedData` (page 2-57) which returns the tag whose index you specify.

Any locale object originally contained in a file has at least one associated-data tag, called the file object tag, that identifies its associated data as the file specification reference.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetIndexedAssociatedData

Given an index into the table of associated data for the specified locale object, returns the associated data, its size, and its tag.

```
OSStatus GetIndexedAssociatedData (
                    LocaleObjectRef objectRef,
                    LocaleObjectTagIndex tagIndex,
                    LocaleDataTag *tag,
                    const void **associatedDataPtr,
                    ByteCount *size);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose associated data you want to obtain.

`tagIndex` A zero-based index that refers to the tag whose data you want. You use `CountLocaleObjectAssociatedDataTags` (page 2-56) to obtain the total number of tags to use as the count.

`tag` A pointer to a value of type `LocaleDataTag` (page 2-19). On output, this value contains an associated data tag identifying the data whose index you specified. If you don't want the function to return an associated-data tag, pass in `NULL` for this parameter.

`associatedDataPtr`
A pointer to data associated with a locale object. On return, this pointer refers to the associated data whose index you specified as the `tagIndex` parameter. The pointer refers to read-only memory. If you attempt to write to it, you will cause an access fault to occur. If you don't want the function to return a pointer to the associated data, pass in `NULL` for this parameter.

`size` A pointer to a value of type `ByteCount`. On output, this value contains the size in bytes of the associated data. If you don't want the function to return the size of the associated data, pass in `NULL` for this parameter.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleObjectFSObjectRef

Returns the file specification object reference for the file that originally contained the locale object.

```
OSStatus GetLocaleObjectFSObjectRef (
                    LocaleObjectRef objectRef,
                    FSObjectRef *fileRef);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose associated file you want to obtain.

`fileRef` A pointer to a file specification object reference. On output, this file specification object reference identifies the file that originally contained the locale object. The file might contain the data for the locale object, for example, if the data is an input method or other service not stored in the locale database. See the File Manager for information on file specification object references. The file specification object reference is valid within the current process only.

function result A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

At system startup, the Locale Object Manager adds all locale objects that exist within files stored in the Locales folder to the locale database. During this process, the Locale Object Manager stores a permanent reference to the file system object specification for the file that originally contained the locale object along with the locale object in the database. The data that a locale object provides can be stored with the locale object in the database or it can remain in the data fork of the locale object's original file. For example, an input method implemented as a (System Object Module) SOM object would reside in the data fork of the locale object's original file. The locale object representing the input method would be installed in the locale database; the locale object would contain the file system object specification for the locale's original file in which the input method, itself, is stored.

The `GetLocaleObjectFSObjectRef` function is used internally by other Mac OS 8 Managers. However it is available for your use as well.

The Text Services Manager, one of the Mac OS 8 managers that uses this function, might create a locale iterator reference (page 2-6) containing function and language attributes, then search the locale database using the iterator to look for services that match the specified values. In the case of a SOM-based text service, for example, the class for the text service would be stored in the data portion of the locale object—not in the locale object's original file—and the locale object would contain a function attribute describing the service's use and a language attribute telling the language for which it is localized. When the Text Services Manager obtained the reference to the locale object for the service it was searching for, it would pass the reference to `GetLocaleObjectFSObjectRef` to obtain the file specification for the file containing the service. The Text Services Manager could then use the file specification to load the CFM (Code Fragment Manager) library and use the class name stored as the locale object's data to instantiate the SOM object service.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Creating and Obtaining a Locale Identifier

A locale identifier is a packed value containing packed language and region codes, and a customization code indicating whether the locale is a customized version of a standard locale. The Locale Object Manager provides these functions for creating and obtaining locale identifiers:

- `CreateLocaleIdentifier` creates and returns a locale identifier based on information you provide.
- `GetSystemLocaleIdentifier` returns the locale identifier for the system locale.

- `GetCurrentProcessLocaleIdentifier` returns the locale identifier for your application's current process.
- `GetLocaleRefLocaleIdentifier` returns the locale identifier for the locale whose reference you provide.
- `GetFirstLocale` returns the locale identifier of the first locale in the database.
- `GetNextLocale` returns the locale identifier for the next locale in the database that follows the locale whose reference and identifier you provide.

CreateLocaleIdentifier

Creates and returns a locale identifier containing the language and region you specify.

```
LocaleIdentifier CreateLocaleIdentifier(LocaleLanguageCode language,
                    LocaleRegionCode region,
                    LocaleCustomizationCode customization);
```

`language` A locale language code (page 2-17) that identifies the language of this locale. If you do not know the primary language for the locale or you do not want to specify a particular language, use the `kLocaleLanguageWildCard` constant (page 2-17) to specify any language.

`region` A locale region code (page 2-18) that identifies the region of this locale. If you do not know the region code for the locale or you do not want to specify a particular region, use the `kLocaleRegionWildCard` constant (page 2-17) to specify any region.

`customization` A customization code set internally by the Locale Object Manager to indicate that the locale is a customized version of a standard system locale. Because this value is set internally by the Locale Object Manager, you should always specify the `kLocaleCustomizationWildCard` constant (page 2-19) for this parameter.

function result A locale identifier (page 2-16) that contains the information you provide to identify a locale.

DISCUSSION

A locale identifier contains a language code, a region code, and a customization code for a locale. You can use `CreateLocaleIdentifier` to create and obtain a locale identifier for a locale that is not in the locale database, that is, for a locale that is not installed on the system. The following scenario illustrates why you might want to create a locale identifier for a locale not resident on the system.

Suppose your application allows a user to label text with language and region attributes; the user directs you to treat a portion of text as French-Canadian, but the locale database does not contain a French-Canadian locale. In response, you would call `CreateLocaleIdentifier`, specifying the appropriate language and region codes for a French-Canadian locale. Then, your application would label the text with the Apple Roman (MacRoman) text encoding along with the locale identifier you created.

You should always specify the wildcard constant `kLocaleCustomizationWildCard` as the value of the `customization` parameter when creating a locale identifier because the Locale Object Manager assigns this code internally only to customized versions of a locale. If you want to know whether a specific locale is a custom one, you can use `GetLocaleCustomization` (page 2-70).

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetSystemLocaleIdentifier

Returns the locale identifier of the system locale.

```
LocaleIdentifier GetSystemLocaleIdentifier (void);
```

function result A locale identifier (page 2-16) that contains the primary language and region codes of the locale used for the system and a customization code if the system locale has been customized.

DISCUSSION

At system startup, the Locale Object Manager establishes the default system locale based on the language and region for which the system is localized. You can use `GetSystemLocaleIdentifier` to obtain the locale identifier for the current system locale. Using the locale identifier that `GetSystemLocaleIdentifier` returns, you can call `GetLocaleLanguage` (page 2-68) and `GetLocaleRegion` (page 2-69) if you want to know the language and region for which the system is localized and `GetLocaleCustomization` (page 2-70) if you want to know if the system locale has been customized in any way.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

You can use the `GetSystemDefaultLocaleRef` function (page 2-25) to obtain a reference to the default system locale.

GetCurrentProcessLocaleIdentifier

Returns the locale identifier of the locale used for the current process.

```
LocaleIdentifier GetCurrentProcessLocaleIdentifier (LocaleRef locale);
```

function result A locale identifier (page 2-16) that contains the primary language and region codes of the locale used for the current process, that is, the current CFM (Code Fragment Manager) context, and a customization code if the locale has been customized.

DISCUSSION

At system startup, the Locale Object Manager establishes the default system locale based on the language and region for which the system is localized. The default system locale becomes the default locale for your application, that is, for the current process, unless you change the locale for the current process by calling the `SetCurrentProcessLocale` function (page 2-26).

You can use `GetCurrentProcessLocaleIdentifier` to obtain the locale identifier for the current process locale. Using the locale identifier that `GetCurrentProcessLocaleIdentifier` returns, you can call `GetLocaleLanguage` (page 2-68) and `GetLocaleRegion` (page 2-69) if you want to know the primary language and region used for the current process and `GetLocaleCustomization` (page 2-70) if you want to know if the current process locale has been customized in any way.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetLocaleRefLocaleIdentifier

Returns the locale identifier for the locale whose reference you provide.

```
LocaleIdentifier GetLocaleRefLocaleIdentifier (LocaleRef locale);
```

`locale` A locale reference (page 2-5) to the locale whose locale identifier you want to obtain.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

Using the locale identifier that `GetLocaleRefLocaleIdentifier` returns, you can call `GetLocaleLanguage` (page 2-68) and `GetLocaleRegion` (page 2-69) if you want to know the primary language and region of the locale and `GetLocaleCustomization` (page 2-70) if you want to know if the locale has been customized in any way.

GetFirstLocale

Returns the locale reference and locale identifier of the first locale in the database.

```
OSStatus GetFirstLocale (LocaleRef  *locale,
                         LocaleIdentifier *localeID);
```

`locale`	A pointer to a locale reference (page 2-5). On output, this pointer refers to the locale reference for the first locale in the database.
`localeID`	A pointer to a locale identifier (page 2-16). On output, the pointer refers to the locale identifier for the first locale in the database.
function result	A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can use `GetFirstLocale` to begin an iteration through the database to obtain the locale reference and locale identifier of each locale in succession. After you obtain the locale reference of the first locale, you can pass it to the `GetNextLocale` function (page 2-66) to obtain the locale reference and locale identifier for the next locale in the database, and so on, calling `GetNextLocale` from within a loop to obtain as many sets of information as you require.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

GetNextLocale

Returns the locale reference and locale identifier for the next locale in the database that follows the locale whose reference and identifier you provide.

```
OSStatus GetNextLocale (LocaleRef  *locale,
                        LocaleIdentifier *localeID);
```

`locale`	A pointer to a locale reference (page 2-5). On input, this pointer refers to the locale reference for the locale preceding the one for which you want information. You can supply the pointer to the locale reference returned by the `GetFirstLocale` function (page 2-65). On output, this pointer refers to the locale reference for the next locale in the database.
`localeID`	A pointer to a locale identifier (page 2-16). On input, this pointer refers to the locale identifier for the locale preceding the one for which you want information. You can supply the pointer to the locale iterator returned by `GetFirstLocale`. On output, the pointer refers to the locale identifier for the next locale in the database that follows the one whose reference you supplied in the `locale` parameter.
function result	A result code. See "Locale Object Manager Result Codes" (page 2-72).

DISCUSSION

You can use `GetNextLocale` from within a loop to iterate through the database obtaining the locale references and locale identifiers for all locales or as many in succession as you require. You pass `GetNextLocale` the locale reference and identifier for the locale where you want the process to begin, and it returns the locale reference and locale identifier for the next locale in the database. To begin collecting locale references and identifiers starting with the first locale in the database, call `GetFirstLocale` (page 2-65) outside the loop before you call `GetNextLocale`, passing to `GetNextLocale` the locale reference and identifier returned by `GetFirstLocale`.

Although `GetFirstLocale` and `GetNextLocale` are meant to be used together, you can call `GetNextLocale` passing it the locale reference and identifier for any locale to begin the process from that locale. To obtain a locale reference-identifier set to pass to `GetNextLocale`, you can use `GetLocaleReference` (page 2-23) or any of the functions that return a reference and `GetLocaleRefLocaleIdentifier` (page 2-65) or any of the functions that return an identifier.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

Obtaining Locale Identifier Information

You can obtain the values stored in a locale identifier. To obtain the language code, use the `GetLocaleLanguage` function. To obtain the region code, use the `GetLocaleRegion` function. To obtain the customization information, use the `GetLocaleCustomization` function.

GetLocaleLanguage

Returns the language code for the locale whose locale identifier you supply.

```
LocaleLanguageCode GetLocaleLanguage (LocaleIdentifier identifier);
```

`identifier` A locale identifier (page 2-16) that specifies a particular locale.

function result

The language code representing the language for the locale that you identified in the `identifier` parameter.

DISCUSSION

The `GetLocaleLanguage` function returns the language code for the primary language of the locale whose identifier you specify. This value is one of the language codes defined by the International Standards Organization (ISO) in the "Code For the Representation of Names of Languages, alpha-3 code" dated

December 16, 1991 (ISO CD 639/2 draft proposal). Constants defined for these codes are included as comments in the `TextCommon.h` header file.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain a locale identifier for locales that reside in the database, you can use any of the functions provided by the Locale Object Manager to return locale identifiers (page 2-63) on the current system.

GetLocaleRegion

Returns the region for the locale whose locale identifier you supply.

```
LocaleRegionCode GetLocaleRegion (LocaleIdentifier identifier);
```

`identifier` A locale identifier (page 2-16) that specifies a particular locale.

function result The region code representing the region for the locale that you identified in the `identifier` parameter.

DISCUSSION

The `GetLocaleRegion` function returns the region code for the locale whose identifier you specify. This value is one of the region codes defined by the International Standards Organization (ISO) in the "Code For the Representation of Names of Languages, alpha-3 code" dated December 16,

1991 (ISO CD 639/2 draft proposal). Constants defined for these codes are included as comments in the `TextCommon.h` header file.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain a locale identifier for locales that reside in the database, you can use any of the functions provided by the Locale Object Manager to return locale identifiers (page 2-63) on the current system.

GetLocaleCustomization

Returns the customization code, if one exists, for the locale whose locale identifier you supply.

```
LocaleCustomizationCode GetLocaleCustomization (
                    LocaleIdentifier identifier);
```

`identifier` A locale identifier (page 2-16) that specifies a particular locale.

function result The customization code for the locale that you identified in the `identifier` parameter. If the locale has not been customized, the Locale Object Manager returns `kLocaleIdentifierWildCard`.

DISCUSSION

You can use `GetLocaleCustomization` to determine if the locale whose identifier you specify has been customized. The Locale Object Manager creates a custom locale based on a locale that exists in the locale database when some aspect of the original locale is changed. The Locale Object Manager assigns a customization code to the locale identifier for the new version of the locale. For example, if a French-Canadian locale is modified in some way—suppose any of the default values for the locale, such as a number separator, have been changed—the Locale Object Manager would create a new custom version of the locale and assign it a customization code.

EXECUTION ENVIRONMENT

Reentrant?	Call at secondary interrupt level?	Call at hardware interrupt level?
Yes	No	No

CALLING RESTRICTIONS

This function cannot be called by hardware interrupt handlers or secondary interrupt handlers.

SEE ALSO

To obtain a locale identifier for locales that reside in the database, you can use any of the functions provided by the Locale Object Manager to return locale identifiers (page 2-63) on the current system.

Determining Where a Locale Object Exists in Memory

GetLocaleObjectMemoryContext

Identifies where in memory the specified locale object resides—whether in system-wide memory or your application's per-process heap.

```
OSStatus GetLocaleObjectMemoryContext (
                    LocaleObjectRef objectRef,
                    LocaleObjectContext *context);
```

`objectRef` A locale object reference (page 2-8) to the locale object whose location in memory you want to determine.

`context` A pointer to a value of type `LocaleObjectContext` (page 2-21). On output, the value pointed to specifies where the locale object resides in memory—whether in system-wide memory (`kLocaleObjectIsGlobal`) or your application's per-process heap (`kLocaleObjectIsLocal`).

Locale Object Manager Result Codes

Many of the Locale Object Manager functions return result codes. The various result codes specific to the Locale Object Manager are listed here. In addition, Locale Object Manager functions may return other system-related result codes.

`localeNotFoundErr`	–30001	Database does not contain the specified locale
`localeObjectAttributeNotAvailErr`	–30002	Specified attribute was not found
`localeObjectNoNamesTableErr`	–30005	Specified locale object does not include a names table
`localeBadReferenceErr`	–30006	Specified locale reference is invalid.

`localeObjectNotFoundErr`	–30007	Database does not contain the specified locale object
`localeObjectInvalidReferenceErr`	–30008	Specified locale object reference is invalid
`localeObjectItemFoundIsLastErr`	–30009	Returned item is the last object in the database. You must reset the iterator to continue the search.
`localeObjectNameAttributeConflictErr`	–30010	Duplicate. A locale object having the same locale object key name and attributes as those specified for the new one already exists in the database.
`localeObjectInvalidIteratorErr`	–30020	Specified iterator is invalid
`localeObjectNoNameErr`	–30021	There is no locale name or name ID that corresponds to the name ID you specified.
`localeObjectTagDataNotFoundErr`	–30022	There is no associated data or associated-data tag corresponding to the tag you specified.
`localeObjectCannotDeleteSystemObjectErr`	–30023	Object specified for deletion is a system object. Your application cannot delete a system object from the database.

`localeDuplicateErr`	–30025	Duplicate locale.
`localeObjectDefaultValueNotAvailableErr`	–30026	Default value for which you specified a name ID does not exist.
`localeNoAssociatedDataTagsErr`	–30027	Specified locale has no associated data.

Glossary

locale A loose collection of data, organized as locale objects, that establishes cultural preferences and characterizes the behavior of text-related processes for the locale. A locale contains locale objects whose data pertains to the culture represented by the primary language and region of the locale, but it can also contain other kinds of locale objects. For example, a modern Greek locale might have locale objects containing sorting tables for classical Greek—one for Doric Greek and one for Attic Greek—for use by ancient Greek languages scholars.

locale database A database that serves as a repository of international preferences and data organized into sets of information called locale objects that are clustered along cultural lines, each of which composes a locale.

locale identifier A private data structure that contains a language code, a region code, and a customization code for a locale. Every locale in the database has a locale identifier that specifies the primary language and region of the locale and that indicates, through the customization code, whether the locale is a customized version of a standard system locale.

locale iterator A private data structure used to search the locale database iteratively for one or more locale objects that satisfy matching criteria specified when an application creates the locale iterator.

locale object A set of information containing data, pertaining to a specific culture, for a text-related function. A locale object also contains information used to identify the data it provides. The Locale Object Manager creates and installs locale objects in the database at system startup from locale object resources stored in the Locales folder. An application can temporarily add a locale object to the database for its use during its current process.

locale object attribute A name-value pair that serves to classify the data a locale object contains. A locale object includes an attributes table that can include various attributes; this allows the locale object to be categorized along multiple lines so that it can be accessed according to any collection of its qualities at different times.

locale object key name A name, specified in the locale object name table, that the Locale Object Manager uses internally to catalog the locale object in the database. A key name serves as the primary search key for a locale object.

Locale Object Manager A collection of functions that let you search the locale database for specific locale objects, their data, and their defining information. You can also add locale objects to the database and modify existing ones using these functions.

locale object reference A private data structure that refers to a specific locale object. An application passes a locale object reference to Locale Object Manager functions to obtain the data contained in a locale object or to obtain information about a locale object, such as any of the user-displayable names associated with the locale object, the locale object's key name, and any of its attributes.

locale object names Every locale object in the database has associated with it a name table that contains up to six names associated with the locale object. Most of the names in the name table exist so that an application can describe a locale object to a user. For example, a name table can contain the locale object's user-displayable name and the copyright notice. The name table also contains the locale object key name.

locale reference A private data structure that refers to a specific locale belonging to the locale database.

locale object resource A resource of type `'lobj'` containing three required tables—a resource for the data table whose table type is `'data'`, a resource for the name table whose table type is `'name'`, and a resource for the attribute table whose table type is `'attr'`. A locale object resource can optionally contain a resource for the head table whose table type is `'head'` and any other type of table that you define. The Locale Object Manager installs locale objects in the database, creating them from the locale object resources that define them.

region A particular subset of a language. A region can represent a linguistic or cultural entity, not necessarily corresponding to a nation, whose language is different enough from other versions of the same language that it merits a specific localized version of Mac OS 8 system software. For example, U.S. and British are two regional variations that are subsets of the English language.

system locale object A locale object that the Locale Object Manager loads into the locale database at system startup, building the locale object from a resource file of type `'lobj'` that defines it and adding the locale object to the appropriate locale. A system locale object is considered permanently resident in the locale database—that is, an application cannot remove it or permanently modify it.