This chapter is an overview of Macintosh text handling. It is meant to help you get started by introducing the concepts described in detail throughout the rest of *Inside Macintosh: Text.*

The chapter contains four major sections. The first presents high-level concepts, and the following three develop those concepts further and give important programming suggestions and hints:

- "Macintosh Text Overview" summarizes what text means for Macintosh programmers, including how to support text in multiple languages. It concludes with suggestions for planning your application's level of text handling.
- "Writing Systems and Script Systems" surveys the issues that must be addressed by any computer-based text-handling system, and then describes the organization of the Macintosh script management system, the set of software managers and resources that help you support text-handling capabilities across many languages.
- "How Script Systems Work" describes the approach taken by the script management system to provide multi-language capabilities in areas such as text display, text input, and string manipulation.
- "Script Systems in Use" describes how the computer user interacts with script systems, including installing script systems, switching text input and display from one language to another, and controlling script-system configuration.

If you are developing a text-handling application, read this chapter's first section, "Macintosh Text Overview," before reading any other parts of this book. You can then either read the remainder of this chapter before going on, or start immediately on the other chapters, returning to this chapter as needed for further explanation of script-system concepts and for specific programming suggestions. The chapters that are most important for general application development are "TextEdit," "QuickDraw Text," "Font Manager," "Text Utilities," and "Script Manager." The chapters that are most important for applications that use input methods, or for developers of input methods, are "Text Services Manager" and "Dictionary Manager."

If you are developing or modifying a script system, read this chapter completely before turning to other chapters and appendixes. Those that are most important for understanding script-system design are "Script Manager," "Built-in Script Support," "International Resources," and "Keyboard Resources."

Valuable information related to the topics discussed in this chapter can be found in *Guide to Macintosh Software Localization.* That book discusses features of individual script systems and gives specific techniques for software localization.

Macintosh Text Overview

Text handling on the Macintosh is fundamentally different from the way it is approached on some common text-based computer systems. There is no hardware-based character generator to put text on the screen; there is no standard input/output window (and no `Writeline` command) for easy generation of text messages.

To draw any text, you first must create a window to draw in. In that window, you can then draw shapes, including the shapes of letters. See the Window Manager chapter in *Inside Macintosh: Macintosh Toolbox Essentials* for a discussion of how to create a window.

In accepting text input and storing text in memory, you cannot assume any particular hardware (keyboard) configuration, you should not assume a particular language for input, you should not assume that characters are always represented by ASCII codes, and you should not even assume that a single character is always represented by 1 byte of storage.

This section paints a broad picture of how text processing works on the Macintosh, and presents some fundamental terminology. It also introduces script systems and briefly discusses two components of system software of special interest for text processing. The section concludes with suggestions on how to give your application the level of text-handling sophistication it requires.

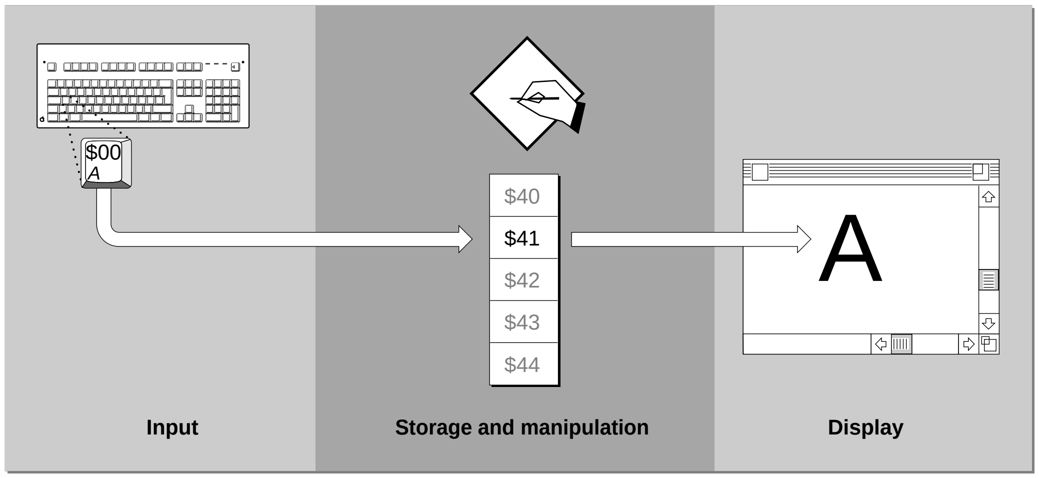
Separation of Tasks

The Macintosh approach separates text handling tasks into three fundamental categories, each relatively independent of the others:

- Text input
- Text storage and manipulation
- Text display

See Figure 1-1. In the discussions in this chapter and elsewhere in this book, keep in mind which category of task is under discussion, to avoid misunderstanding.

Figure 1-1 Separation of input, storage, and display in Macintosh text handling



Through text input, your application obtains representations of text characters. It starts with the user pressing keys on a keyboard. Text input is aided by specialized parts of system software that allow input of text in many languages. Text input is completely independent of text display.

Your application stores character representations in memory, as numeric codes. The main focus of your application is on storing, tracking the characteristics of, and manipulating these codes in memory. How those codes got into memory, and how you will display or print them, are mostly separate issues. For much of your processing, you will be concerned with codes in a memory buffer, rather than keypresses on a keyboard or pixel locations on a screen. The system software has many routines that aid in manipulating text of many languages in memory.

Note
In Figure 1-1 and throughout this book, text in computer memory is drawn as a vertical table of codes, representing sequential (downward) storage of text characters in a buffer. Some diagrams also include byte offsets in the buffer, and even miniature representations of the characters themselves in a given language. See, for example, Figure 1-3 on page 1-8. ◆

Though text display, your application makes visible the characters it has stored and manipulated in memory. The end result of the display process is a sequence of text shapes drawn on a display device. As is shown later in this chapter, the displayed form of text often has a complex relationship to the way it is stored. In most cases you can consider text display as an independent task, handled in large part by system software, that you call after you have finished receiving, storing, or otherwise processing characters in memory. It is only during display, for example, that the concept of a font has meaning. (Preparation for text display, such as width measurement and line-breaking, falls on the boundary between storage and display, and is in general a cooperative effort between your application and system software.)

If your application is a word processor that is drawing characters to the screen as the user enters them, all three of these tasks are closely coupled in time. Nevertheless, they are still independent of each other and can be understood best as separate processes.

Text Is Graphics

Your application draws graphic shapes on the Macintosh screen by making calls to QuickDraw, the graphics manager of Macintosh system software. The graphics components of QuickDraw are described in the chapter “QuickDraw,” in *Inside Macintosh: Imaging*.

Drawing text is fundamentally the same as drawing graphics. The application makes QuickDraw calls to write text to the screen or to a printer. Those parts of QuickDraw that are concerned specifically with drawing text are documented in the chapter “QuickDraw Text” in this book.

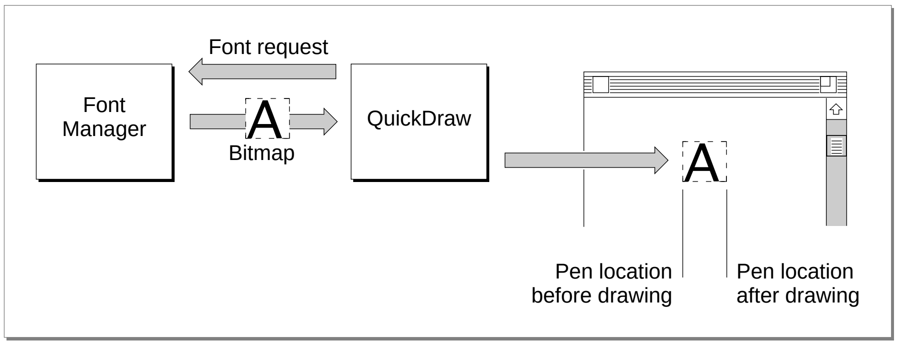
When QuickDraw draws text, it places bitmapped shapes on a display device. Those shapes are the forms of individual letters in a particular font. A **font** is a resource that contains a complete set of character representations in a particular typeface, such as Times® or Geneva. Without a font, QuickDraw cannot draw text.

When you ask QuickDraw to draw text, it draws it according to the settings of the window (specifically, of the current **graphics port** record) that you are drawing into. The text's screen location, font, size, color, and style are all implied by the current state of the graphics port; they are not explicit parameters of your text-drawing call.

For example, when QuickDraw draws a character, it draws it at the current **pen position,** the screen position at which drawing occurs, in the current window. The character's origin (usually its left edge) is placed with respect to that location, with the rest of the character extending to the right of the origin. After drawing, QuickDraw automatically updates the pen location by the width of the character, so that the next character drawn will be automatically placed the correct distance to the right of the first. See Figure 1-2.

Likewise, when QuickDraw draws a string of text, it keeps advancing the pen location as it draws, so that the current location ends up at the right end of the string. This left-to-right orientation of QuickDraw is fundamental, and applies whether or not the text being drawn is meant to be read left-to-right (such as English) or right-to-left (such as Arabic).

Figure 1-2 How QuickDraw draws text



QuickDraw's text-measuring capabilities are as important as its drawing capabilities. In many cases, before you draw a line of text, you first need to know its length in pixels, so that you can correctly place it on the screen and be assured that it does not overrun its allotted space. Pixels are screen dots, and are nominally equal to one point, or approximately 1/72 inch, in size. You often make two sets of QuickDraw calls when drawing a string; the first to measure it, and the second to actually draw it.

The **Font Manager** supports QuickDraw by providing the character bitmaps that QuickDraw needs, in the typefaces, sizes, and **styles** (such as bold or italic) that QuickDraw requests. The Font Manager keeps track of all fonts available to an application. If QuickDraw requests a typeface that is not represented in the available fonts, the Font Manager substitutes one that is; if QuickDraw requests a size that is not available, the Font Manager scales an available size and returns the bitmaps to QuickDraw; if QuickDraw requests a style that is not available, the Font Manager returns an unstyled set of bitmaps and QuickDraw applies a style to them (by slanting for italic, or darkening extra pixels for boldface, and so on). In general, the Font Manager does the calculations and creates the bitmaps; QuickDraw transfers those bitmaps to the screen.

Fonts are strongly language-dependent. A font is the manifestation of the character set—the body of meaningful characters—of a language or group of languages, called a **writing system.** Fonts also implement additional symbols and forms, such as ligatures, needed by that writing system. The Font Manager provides for fonts in many writing systems; fonts are identified by a numbering scheme with which the writing system of a font can be determined from its number.

Macintosh fonts come in two basic kinds: **bitmapped** and **outline** (such as TrueType). Each bitmapped font is a set of character bitmaps of a given typeface in a single size; each outline font is a set of templates from which bitmaps of any size can be generated. All Macintosh text-handling routines work with both types of fonts.

Fonts can also be classified by the sizes of the character sets they implement. The typical Macintosh fonts, suitable for most languages of the world, are called 1-byte fonts; each contains fewer than 256 characters. Fonts for some East Asian languages, however, need thousands of characters; they are called 2-byte fonts. The Macintosh text-handling routines can work with both 1-byte and 2-byte fonts, although special techniques may be required for character handling with 2-byte fonts. Bitmapped and outline fonts are described in the chapter "Font Manager" in this book and in *TrueType Font Format Specification*, available from APDA. For more information on how fonts are used on the Macintosh, see "Fonts" beginning on page 1-44, and "Font Handling" beginning on page 1-60.

The text measuring and drawing routines in QuickDraw and the Text Utilities operate under certain assumptions, based principally on the fact that Macintosh system software was originally developed for the left-to-right Roman writing system of the English language, and that the system software provides line-layout, but not page-layout, capabilities. Remember these points:

- QuickDraw draws all text from left to right. Whether your text has a left-to-right or right-to-left **line direction**—the direction in which the text is read—QuickDraw places its left edge at the current location in your window and draws its characters in order from the leftmost to the rightmost character. QuickDraw and the Text Utilities provide routines that allow you to order and draw your text properly regardless of its line direction or directions.
- On a line of text, screen position is in terms of pixel offset from the *left* edge of the text-drawing area, regardless of the line direction of the text being drawn.
- The text-measuring routines in this book help you calculate and lay out individual lines; it is up to you to track where a line starts, both in terms of vertical screen position and in terms of offset in your text run.

Characters, Glyphs, Character Codes, and Bytes

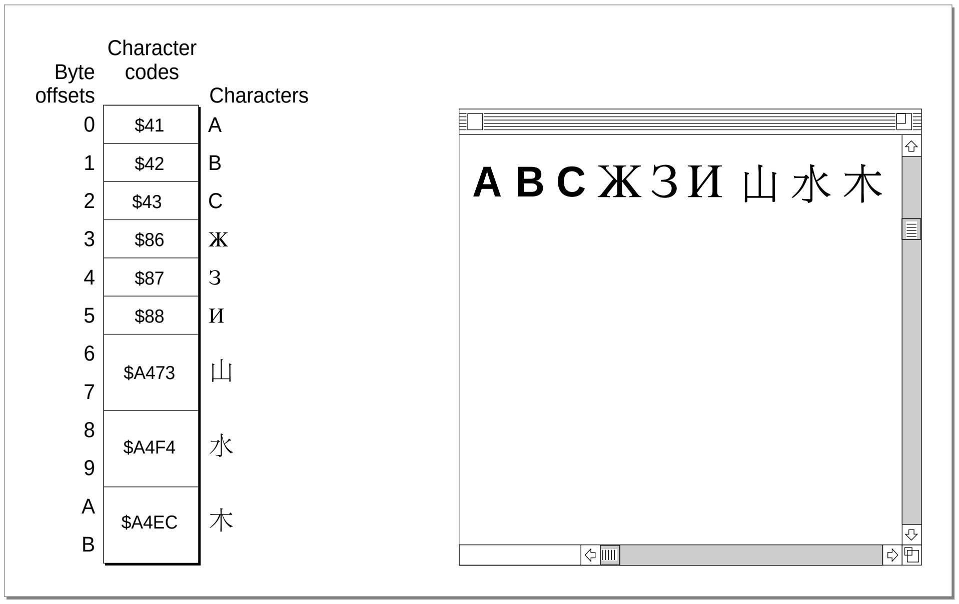
In memory, applications store text as numerical representations of characters. On the screen, QuickDraw draws text as bitmapped representations of those characters, generated from a particular font. To clarify how numbers in memory are converted to letters on the screen, keep the following terms in mind. See also Figure 1-3.

A writing system's alphabet, numbers, punctuation, and other writing marks consist of **characters.** A character is a symbolic representation of an element of a writing system; it is the concept of, for example, "lowercase a" or "the number 3". It is an abstract object, defined by custom in its own language.

As soon as you write a character, however, it is no longer abstract but concrete. The exact shape by which a character is represented is called a **glyph.** A font, then, is a collection of glyphs, all of similar design, that constitute one way to represent the characters of the language. The "characters" that QuickDraw places on the screen are really glyphs.

In memory, text is stored as **character codes,** where each code is a number that defines a particular (abstract) character. The "characters" that an application reads into or out of a buffer, sorts, and searches for are really numeric codes. One purpose of a Macintosh font is to provide glyphs that the system software can associate with character codes; different fonts for the same language will typically have different glyphs, all representing the same character, for a specific character code. Thus no matter which font you use, an English "C" is always a "C" (character code $43), though it may be Garamond or Chicago font, italic or bold style, and 7 points or 72 points in size. (Note that fonts in certain languages may have more than one glyph per character, and may have special glyphs for various combinations of characters.)

Figure 1-3 Bytes, character codes, characters, and glyphs



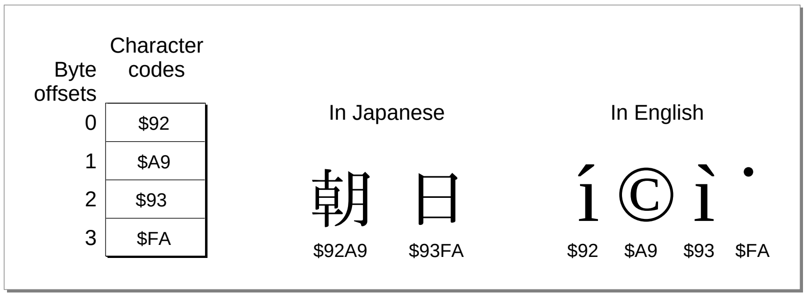
In computer memory, 1 byte (8 bits) is commonly used to store a single character code. For most languages that is sufficient: the standard **ASCII character set** (also called **low ASCII**) requires only 7 bits per character code, and the Apple Standard Roman character set (an extended ASCII character set derived from the original Macintosh character set) requires only 1 byte per character code. In many other languages, such as Russian, Arabic, and Thai, each character code is also 1 byte in size. But in some East Asian languages such as Japanese, Chinese, and Korean, the character set is so large that most character codes must be 2 bytes long. Macintosh system software provides routines to help you recognize and manipulate 2-byte characters; if your application is to be useful throughout the world, *you must be prepared to deal with 2-byte characters.*

The left side of Figure 1-3 shows a portion of a text buffer in memory. Byte offsets into the buffer are shown down the left side of the column. The character codes the buffer contains are shown within the column; note that some codes are a single byte, whereas others are 2 bytes in size. (For clarity, miniature representations of the characters defined by those character codes are shown down the right side of the column.)

The right side of Figure 1-3 shows what happens when QuickDraw draws the contents of the buffer. The character codes define which glyphs are placed on the screen, and in what order. The character codes do not define the style or size of the glyphs, however.

Character codes are only numbers; the meaning of each character code is different in different writing systems. In Figure 1-4, for example, the same four bytes are interpreted very differently if they are considered to be two Japanese character codes than if they are considered to be four English (= Roman writing system) character codes.

Figure 1-4 Four bytes displayed in Japanese and in English



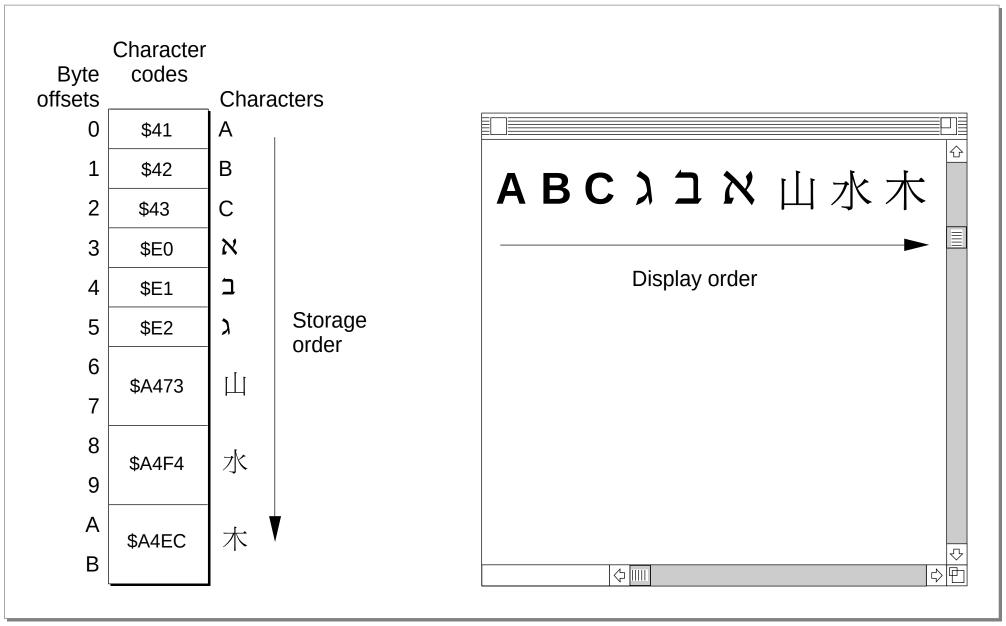
Text Storage

In considering how to store text in buffers, strings, and files, it may be clearer if you understand the assumptions that the Macintosh text managers make about your text-storage method. The discussions throughout this book assume that your text is stored and accessed acording to these conventions:

- Your program stores text as a simple sequence of character codes. The character codes may be 1-byte or 2-byte codes, but there is nothing else in the text stream except for those codes. Using font information that your program stores separately, Script Manager routines can help you determine whether a character is 1 or 2 bytes, and other managers allow you to work with either character size.

- Character location within a text sequence in memory is in terms of byte offset (not character offset) from the beginning of the text. Offset is zero-based; the first byte in the sequence has an offset of 0.
- The **storage order** of your text—the sequence in memory in which the character codes occur—is the same as its logical order. It is the order in which the characters would be read or pronounced in the language of the text. Because text of different languages may be read either left-to-right or right-to-left, storage order is not always the same as the left-to-right **display order** of the text when it is drawn. In Figure 1-5, for example, note that the Hebrew characters are displayed in reverse order from the order in which they are stored.

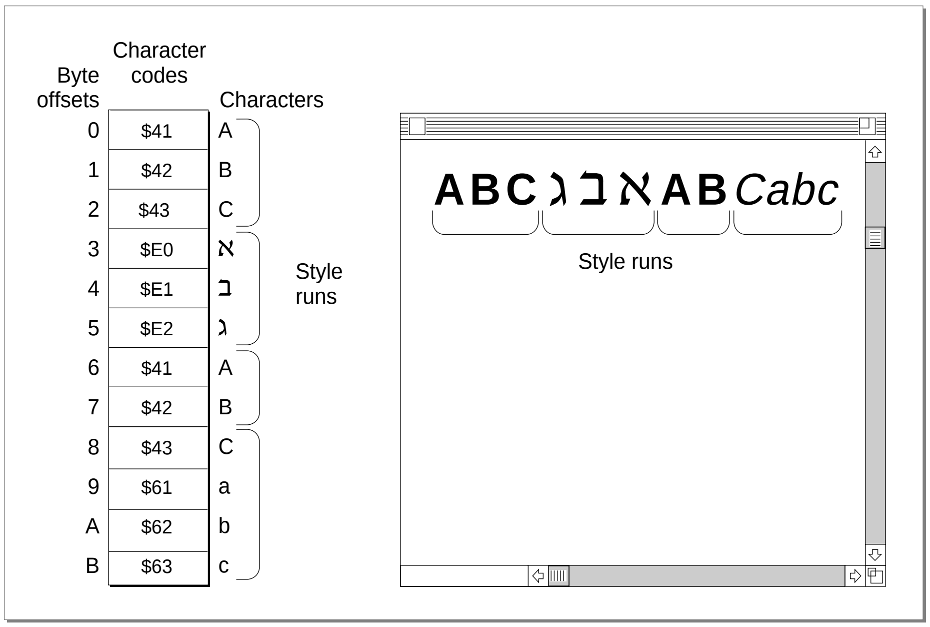
Figure 1-5 Storage order and display order



- All writing-system, font, size, color, and style information about each part of your text is stored separately from the text, and it is your application's responsibility to maintain that information. The text stream itself carries no information about what writing system or font it was created with or is meant to be drawn with; you need to keep track of and supply that information before making a drawing or measuring call.
- Text is divided into **runs.** There are text runs, direction runs, script runs, font runs, and style runs. A **style run** is a continuous sequence of text that is all of the same writing system, font, size, color, style, and scaling factors (if the text is scaled). Figure 1-6 shows four style runs on a single line. Because of the way many drawing and measuring routines work, it is important to track all the individual style runs in

your text. Runs are described in more detail under "Style Runs, Font Runs, Script Runs, Direction Runs" beginning on page 1-70.

Figure 1-6 Style runs in text



- Drawing involves converting character codes in memory to glyphs on the screen. When drawn, some characters in some writing systems change their shape, size, or position depending on their contextual position, that is, on what other characters surround them. See "Contextual Forms and Character Reordering" beginning on page 1-26. Using information in a set of **international resources,** the Macintosh text-measuring and drawing routines can automatically perform these contextual transformations for you.
- For text that is contextual, you do not store the transformed, ready-to-draw version; what you store in memory are the codes for the fundamental characters that make up the text. That makes searching, sorting and other manipulation more straightforward. Each time the text is drawn it is re-transformed as appropriate.

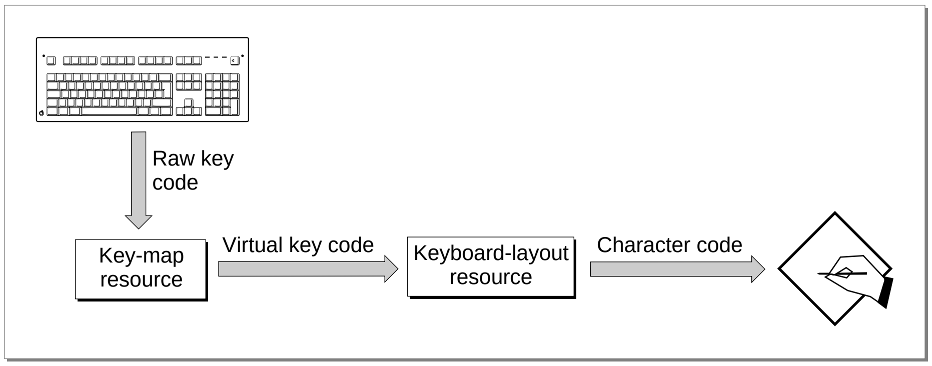
Keyboards and Input Methods

By means of keyboard input, the user can create text that your application stores as character codes and displays as glyphs. At first glance this may seem a difficult task: your application should be able to handle input from at least 13 different hardware types of Apple keyboards, as listed in the appendix "Keyboard Resources" in this book. Furthermore, it must be able to derive the proper character codes for any writing system from each of the keyboards and recognize the states of the **modifier keys** (Shift, Caps Lock, Command, Option, and Control).

The system software and the **keyboard resources** make this relatively easy for you. The Event Manager uses the keyboard resources to convert keypresses into the correct character codes for the current writing system, for whatever keyboard is used. Your application receives the codes directly and needn't keep track of the specific keyboard in use.

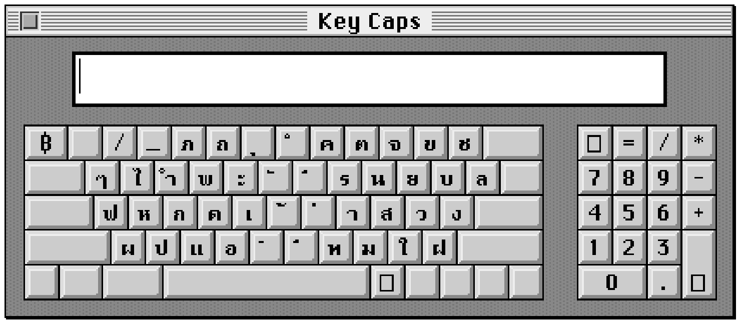
Figure 1-7 is a simplified view of **key translation,** the process by which character codes are generated. Each keyboard has a particular physical arrangement of keys, and each keypress generates a value called a **raw key code,** which indicates which key was pressed. The keyboard driver that handles the keypress uses the **key-map resource** to map these raw key codes to keyboard-independent **virtual key codes.** It then uses the Event Manager and the **keyboard-layout resource** to convert a virtual key code into a character code, and passes it to your application in the event record generated by the keypress. See "Keyboards and Key Translation" beginning on page 1-87 for a more complete description of key translation; see the Event Manager chapter in *Inside Macintosh: Macintosh Toolbox Essentials* for a description of events and event records.

Figure 1-7 Key translation (simplified)



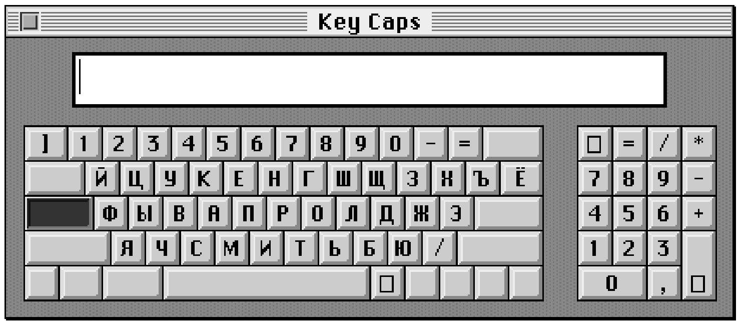
Keyboard layout can be considered the overall relationship between the physical arrangement of keys on a keyboard and the glyphs produced when those keys are pressed. It is what the Key Caps desk accessory shows; see Figure 1-8.

Figure 1-8 Key Caps display of Thai keyboard layout (no modifier keys pressed)



Changing the physical keyboard, changing the keyboard-layout resource, pressing modifier keys, and changing the font can all change the relationship between keypresses and glyphs. Figure 1-9 is a Key Caps display for the same physical keyboard as that in Figure 1-8, but the writing system has been changed from Thai to Cyrillic. For the purposes of this book, however, the keyboard-layout resource is the critical item in determining keyboard layout; *changing the keyboard layout means changing the keyboard-layout resource.* Because keyboard layouts are independent of the physical keyboard attached to the computer, your application has the flexibility of changing text input from one writing system to another by simply using a different keyboard-layout resource.

Figure 1-9 Key Caps display of Cyrillic keyboard layout (Caps Lock key pressed)



For languages with large character sets, it is impractical to manufacture keyboards with keys for every possible character. In such a case, it is usually the job of an input method, working in conjunction with a keyboard, to handle text input. An **input method** is a software module, often independent of the application it serves, that converts character codes that can be entered from the keyboard into character codes that cannot. Japanese and Chinese input methods commonly display a small window, into which the user types a sequence of phonetic characters; the input method converts them into one or more ideographic character codes and sends them to the application. A more sophisticated input method is **inline input,** in which entry and conversion of text occur directly in the window of the text document being edited. See "Input Methods" beginning on page 1-91 and the chapter "Text Services Manager" in this book for more information on input methods and inline input.

In most cases, your application does not need to do anything special related to keyboard input. You can use the character codes returned by the Event Manager function `WaitNextEvent`—whether generated directly from keypresses or through an input method—and handle the text appropriately for the language being used for input. Remember, however, that keyboard input is independent of text display; it is your responsibility to keep the two synchronized when necessary. If the user switches language for text input, you must switch the language for text display accordingly. The Font Manager and Script Manager provide routines that help you with that; see "Font and Keyboard Synchronization" beginning on page 1-90, and further discussion in the chapter "Script Manager" in this book.

Writing Systems and Script Systems

Localization is the process of adapting software to local use. When a version of Macintosh system software is created for a particular country or region, its text strings usually must be translated and it must support the writing system of that region. To facilitate the localization of Macintosh system software around the world, much of Macintosh text-handling is concerned with proper presentation in multiple languages. Macintosh computers are sold worldwide, and Macintosh system software is currently available in over 30 localized versions, allowing computer users in many parts of the world to use the Macintosh in their native languages. Macintosh system software likewise provides your application with the capability of simultaneously supporting multiple writing systems.

IMPORTANT
Even if you do not plan to localize your application, it should still support multiple writing systems. Users in your own target region may have capability for more than one writing system on their computers, and may want your application to support that capability. ▲

In this book, a **writing system** denotes a method used to depict words visually. It consists of a character set and a set of rules for displaying, ordering, and formatting those characters. Writing systems can differ in **line direction,** the direction in which their characters are read; the size of the character set used to represent the writing system; and whether or not they are contextual—whether a character changes its form depending on its position relative to other characters. Writing systems have specific requirements for text display, text editing, character set, and fonts. A writing system, of which one example is Roman, can serve more than one language, of which two examples are French and Spanish. A single language such as French can have regional variations with slightly different requirements, such as Swiss French and Canadian French. Writing systems and their features are described under "Features of the World's Writing Systems" beginning on page 1-21.

On the Macintosh computer, a **script system** (or **script** for short) is a collection of resources that provides for the representation of a particular writing system. A script's **keyboard resources** define the character codes and keyboard layout for the writing system, and its **international resources** provide a host of formatting and ordering rules for the writing system. A script system requires one or more fonts designed specifically for the writing system. The script system is accessed through the Script Manager, the Text Utilities, the script extensions WorldScript I and WorldScript II, and the other text-related software managers described in this book. Together, these software components make up the **Macintosh script management system.** The files, managers, and resources that make up the script management system are described under "Components of the Macintosh Script Management System" beginning on page 1-35.

A script system on the Macintosh is identified primarily by number, its **script code.** And just as writing systems can serve several languages, script systems can have variations for different languages, specified by **language code.** Each language code "belongs" to a particular script code. Regional variations can also be reflected in script systems, by **region code.** Each region code "belongs" to a particular language code. See, for example, Figure 1-34 on page 1-49.

More than one script system may be **enabled**, or present and available, on the Macintosh. Script systems may be installed either as **auxiliary scripts,** which just provide writing-system support, or as the *system script,* which affects system defaults such as the default font, keyboard layout, line direction, and so forth, and is typically the writing system used for localized dialog boxes, menus, and alerts. All other scripts are secondary to the system script. The **font script,** also called the *current script,* is the script system currently being used to draw text. The **keyboard script** is the script system currently being used for text input.

The Roman script system is always available, either as the system script or as an auxiliary script. Furthermore, the low-ASCII Roman characters are always available in any script system; they are a standard part of every script system's character set.

Macintosh system software routines that take into account the script system of the text they manipulate are called **script-aware** routines. Likewise, applications that use those routines to properly handle text according to its script system are also called script-aware. Your applications should be script-aware.

More details about script systems and how they work are found under "Components of a Script System" beginning on page 1-40, "How Script Systems Are Classified" beginning on page 1-45, and "How Script Systems Work" beginning on page 1-52.

Macintosh Text Utilities

The Text Utilities are a broad collection of text-manipulation routines provided by Macintosh system software. With Text Utilities calls, you can

- specify strings for various purposes
- sort strings, including strings in any writing system and combinations of strings in different writing systems
- convert case or strip diacritical marks from text for sorting purposes
- format numbers and currency
- format dates and times
- search and replace text
- find word boundaries and line breaks when laying out lines of text

Some Text Utilities routines function with the Roman script system only, but many are script-aware and work properly with all script systems. Script-aware Text Utilities routines rely on a script system's international resources to define the specific behavior in that script system.

The Text Utilities are described in the chapter "Text Utilities" in this book.

TextEdit, a Text-Processing Service

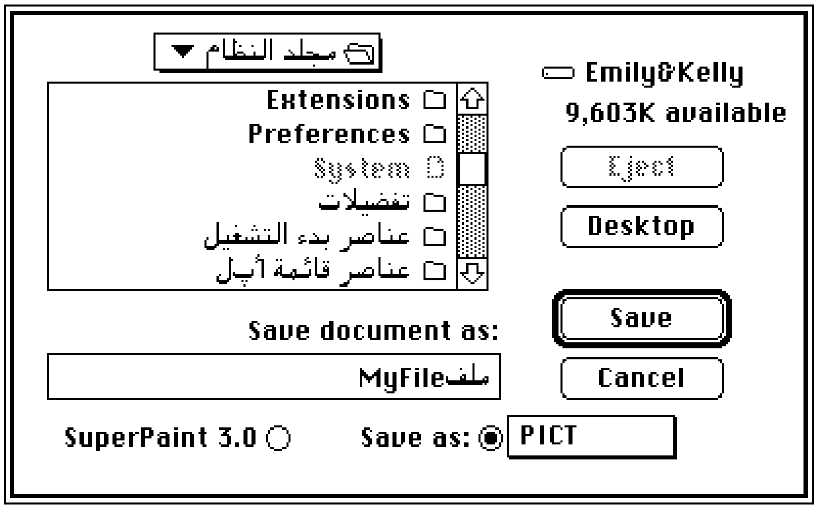
Macintosh system software provides a simple text-processing service, used by the Dialog Manager and other parts of system software, and available for your use also. TextEdit handles certain basic text-handling tasks for small (less than 32 KB) amounts of text.

TextEdit maintains a text buffer, provides line breaks, tracks the selection range and insertion point for text, handles insertions and deletions from the buffer, and tracks style information for all its text. TextEdit formats and draws text properly in multiple styles and different script systems—even multiple scripts on a single line. TextEdit handles mixed-directional text, synchronizes fonts and keyboards, handles 2-byte characters, determines word boundaries, and matches text alignment with line direction. TextEdit even allows you to customize several of its features, such as word selection and text measurement.

If you want multiscript text handling, and you do not need to manipulate large files and do not need formatting other than font styles, TextEdit is a convenient alternative to writing your own text processor. You can use TextEdit at different levels of complexity:

- For the very simplest text handling (in dialog boxes), you needn't even call TextEdit directly. Use the Dialog Manager, which in turn uses TextEdit, to correctly edit and display text in either the system script or Roman script. For example, the Save As dialog box shown in Figure 1-10 handles mixed-directional text (in this case, Arabic) correctly. The Dialog Manager is described in *Inside Macintosh: Macintosh Toolbox Essentials.*

Figure 1-10 TextEdit edits and displays mixed-directional text in a dialog box



- If you simply want to display one or more lines of static (non-editable) text, you can call the TextEdit `TETextBox` procedure. `TETextBox` draws your text at the location you specify with the alignment you specify. You need not make any other TextEdit calls or allocate any data structures if you use `TETextBox`.
- Other than dialog boxes and static text display, if your application requires very basic text handling, in which neither styled text nor multiple fonts are needed (as in many desk accessories), you may need only monostyled TextEdit. You can use monostyled TextEdit with the application font (if you don't allow the user to select a font) or with any single available font (if you do allow user selection) in any version of Roman or non-Roman Macintosh system software.
- If your application requires a somewhat higher level of text handling (allowing the user to set the font, size, and style of text, for example), you can use multistyled TextEdit. You can use multistyled TextEdit with any combination of available fonts, in any version of Roman or non-Roman Macintosh system software.

TextEdit does have limitations; it is not powerful or efficient enough for use as a general text editor. For example, TextEdit

- can only handle up to 32 KB of text
- is not highly optimized for speed
- contains data structures that can be inefficiently large for multistyled text

Nevertheless, TextEdit's convenience and multiscript capabilities make it an attractive alternative to writing your own text processor. TextEdit is described in the chapter "TextEdit" in this book.

Planning Your Text Handling Capabilities

The Macintosh system of text handling—with its graphic approach to text drawing, separation of text storage from text rendering, ability to handle many writing systems, event-controlled text input, large library of utility routines, and availability of a simple text-handling service—is general and powerful. But you may not need all of its power, and the simpler your needs are the less you will have to do to meet them.

It may appear difficult at the outset to generalize your text-handling capabilities so that they can work across all script systems around the world. You may instead wish to customize your application to work with a specific regional variation or script system in a target market that interests you. Either approach is possible; you can use the Macintosh script management system to build in language-independence or language customization, as you wish. There are three general approaches you can take:

- **Globalization** is the preparation of a culturally neutral application that provides the technical underpinnings for script-specific, linguistic, and regional variations, and that is capable of running with any script system. Globalization involves careful design and writing of the application and its textual and graphic resources.
- **Localization** is the adaptation of an application to a particular language or region, to achieve proper formats for dates, times, currency, measurement, calendars, and numbers, proper text sorting, and acceptable forms of other culturally specific material. Localization involves translation of textual resources, modification of graphic resources such as icons, and possibly creation of a customized set of script-system resources. The better globalized an application is, the easier it is to localize.
- **Customization** is the inclusion of script-specific, linguistic, or regional capabilities supporting features that are not otherwise supported by Macintosh system software (for instance, vertical text direction or special underlining modes for the Japanese writing system).

This book supports and describes the process of globalization; it helps you prepare your application to support all writing systems and regions. The process of localization is discussed in *Guide to Macintosh Software Localization*. This book does not discuss customization, beyond the few suggestions presented at the end of this section.

To achieve globalization, localization, or customization, the level of work required is related to the level of text-handling sophistication you need. There are three general levels to consider—rudimentary, moderate, and highly sophisticated.

Rudimentary Text Handling

Rudimentary text handling means that the user either cannot set fonts at all (the lowest level of sophistication) or that the user can set fonts and styles but not alignment (a slightly higher level). In either case large amounts of text and sophisticated formatting are not required.

If your application requires only rudimentary handling, use TextEdit—either directly or through the Dialog Manager—to handle user input and editing. TextEdit exhibits the correct behavior for editing and displaying text in multiple styles and different script systems.

In addition, at an absolute minimum, design your application so that it can display its own Roman text properly when operating with a non-Roman script system. For text in dialog boxes, menus, alert boxes, and so on, if you do not plan to translate the text for localization use only the low-ASCII character codes that are the same on all script systems. High-ASCII character codes may map to incomprehensible characters in another script. The ellipsis in menu items, for example, maps to other characters when displayed in other system scripts. Instead of using the ellipsis, a high-ASCII character code, you can use three periods, a low-ASCII string; the ellipsis is displayed regardless of the system script. (A better approach, however, is to use the script management system to retrieve the appropriate form of the ellipsis character for whatever script system you are running under. See the discussion of retrieving text from tokens in the chapter "Script Manager" in this book.)

Moderate Text Handling

Moderate text-handling sophistication means an application allows users to set font, style, alignment, tabs, writing direction, keyboard, input method, and so forth, across script systems. It handles large amounts of text and offers greater formatting sophistication than TextEdit provides.

The Macintosh script management system and all the text managers documented in this book are designed to support this level of sophistication. You can use these managers and the rest of Macintosh system software to include basic word-processing capabilities in your application, capabilities that work across the entire range of worldwide writing systems supported by Macintosh system software.

Within the range of moderate text handling, the level of complexity is largely a function of the number and types of script systems that are currently enabled. You may wish to structure your application's text-handling algorithms to allow for categories of increasing complexity, based on conditions such as the following four. (The item in parentheses following each condition is a selector or flag that tests for that condition. See the discussion of selectors for Script Manager variables and script variables in the chapter "Script Manager" in this book.)

- Only one script system is present (`smEnabled` = 1). If there is only one script system, it is Roman; you can assume all text-handling follows the built-in Roman rules, and you do not need to account for or test for the script system of any text.
- More than one script system is present (`smEnabled` > 1). You need to track the script system associated with each run of text. You need to use script-aware routines for text handling. You need to synchronize the font script with the keyboard script.

- A bidirectional script system is present (`smBidirect` = `TRUE`). You need to allow for the possibility of right-to-left text, right alignment of text, discontinuous highlighting, and a display order for style runs that is different from storage order. You need to allow for contextual behavior in drawing; you cannot use font width tables for measuring text.
- A 2-byte script system is present (`smDoubleByte` = `TRUE`). You need to allow for the presence of 2-byte character codes in searching, drawing, and line-breaking. You should also support inline input of text whenever the keyboard script is a 2-byte script system.

These probably represent the major divisions in text-handling complexity that you address, although you may want to account for others. For example, you may want to test each individual script system to see if it is contextual (`smsfContext` set), 1-byte (`smsfSingByte` set), or bidirectional (`smScriptRight` = `TRUE`) before deciding how to handle its text.

With the moderate level of text-handling supported by the Macintosh script management system as documented in this book, your application can be powerful enough and general enough for worldwide acceptance.

Sophisticated Text Handling

Highly sophisticated text processing might be employed by a very powerful word processor that works across many script systems. If you write such a program, you will probably need to go beyond the capabilities provided by the Macintosh script management system.

Areas that may need special attention include specialization or customization of delimiters, higher-level grammatical structures, word selection, sorting, arrow keys, and line direction. All of these issues are addressed by the current Macintosh script management sytem, but if your needs go beyond what the system is now capable of, you may need to write your own code to accomplish them. Here are a few examples:

- Your application may require the implementation of functions not supported by the Macintosh script management system but needed by certain languages—for example, text with a vertical line direction.
- Your application may mark text by language, and allow users to limit searching, sorting, or spell-checking to specific languages.
- Your application may want to display characters in a more sophisticated manner than is supported—such as *furigana*, also called *rubi*, a Japanese text display in which small Kana characters are placed adjacent to a Kanji character to indicate its pronunciation or to explain it if it is rare.

If you do write your own code to replace one or more of the Macintosh script management capabilities, make sure you do it in a modular fashion, so that you can work with current Macintosh text managers and also be prepared to take advantage of possible future enhancements to system software.

Writing Systems and Script Systems

The first section of this chapter, "Macintosh Text Overview," has given an overview to all of Macintosh text handling. This section and the rest of the chapter develop many of those concepts in more detail, to give you the background necessary to work with the routines documented throughout the remainder of this book.

This section presents the language features that must be addressed by system software if it is to properly handle the world's writing systems. It also describes the organization of the Macintosh script management system and the structure and classification of the Macintosh script systems that implement that international text handling.

Features of the World's Writing Systems

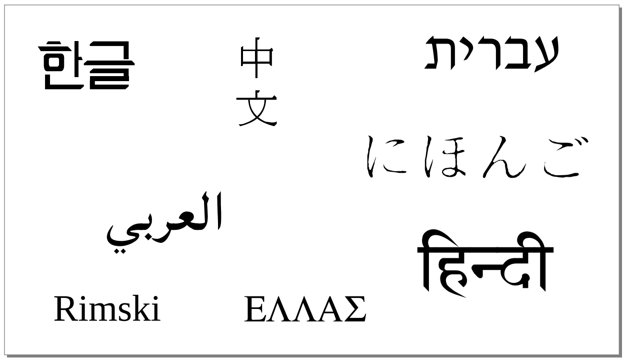
In order to understand the structure and workings of Macintosh text handling, it is useful to first consider the range of text features that need to be represented on the computer. This section presents the principal text-related features, taken from writing systems around the world, that the Macintosh addresses.

A **writing system** is a set of characters and the basic rules for their use in creating a visual depiction of language. There are more than 30 active writing systems in the world today, used to represent the official written languages of one or more regions and countries. Examples of writing systems are Roman, Chinese, Japanese, Hebrew, and Arabic. Color Plate 1 shows the world distribution of some of the principal writing systems.

Each writing system has distinct attributes. Simple systems such as Roman, Greek, and Cyrillic usually have fewer than 200 characters; Japanese, a complex writing system, theoretically contains more than 40,000. Printed Roman characters are relatively independent of each other; Arabic characters change shape depending on the characters that surround them. Some writing systems use spaces to separate words; others do not separate words at all. Some writing systems, such as Japanese, actually include multiple subsystems, each with its own set of characters and rules for how they are combined.

Figure 1-11 shows the names of various languages and regions, written in the appropriate writing system for each language.

Figure 1-11 Writing-system examples



The variety of writing-system attributes presents difficult, though not insurmountable, challenges to their representation on the Macintosh computer. This section discusses the principal attributes that the Macintosh script management system addresses.

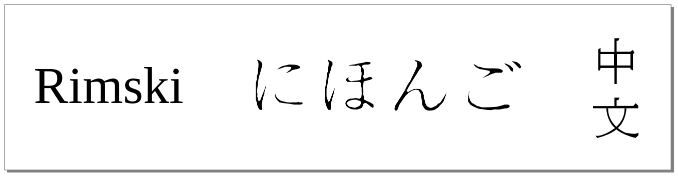
Character Representation

Writing systems differ in the kind and number of characters required to create words as the basic components of language. Some writing systems, such as Roman and Cyrillic, are basically alphabetic: the characters in the writing system symbolize, more or less, the discrete phonemic elements in the languages represented by that writing system. Other writing systems, such as Japanese Kana, are syllabic: the characters stand for syllables in the language.

Some writing systems—namely, Japanese **Kanji**, Chinese **Hanzi**, and Korean **Hanja**—include **ideographic** characters. These characters do not represent pronunciation alone, but are also related to the component meanings of words. A typical character set for ideographic writing systems is quite large, ranging from 7,000 to 30,000 characters.

Figure 1-12 shows examples of alphabetic, syllabic, and ideographic representations of characters.

Figure 1-12 Words with alphabetic, syllabic, and ideographic characters

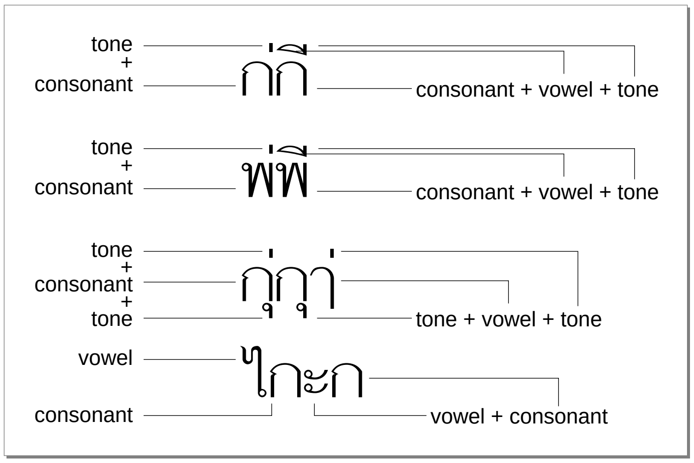


Several writing systems, including Hebrew, Thai, and Korean, contain character clusters. A **character cluster** is a collection of alphabetic characters.

- In some systems, character clusters consist of a principal character plus attachments in memory. For example, in Hebrew, a cluster may be composed of a consonant, a vowel, a dot to soften the pronunciation of the consonant, and a cantillation mark.
- In other systems, character clusters occur as alphabetic blocks made of 2 to 5 component parts. For example, in Korean, consonant and vowel components called Jamo are combined into blocks called Hangul. See Figure 1-39 on page 1-60 for an example. In Thai (as shown in Figure 1-13), consonants are combined with vowel marks and tone marks to make clusters.

On the computer, character clusters pose difficulties in the treatment of word demarcation, the movement of the caret, deletion, and highlighting.

Figure 1-13 Thai character cluster



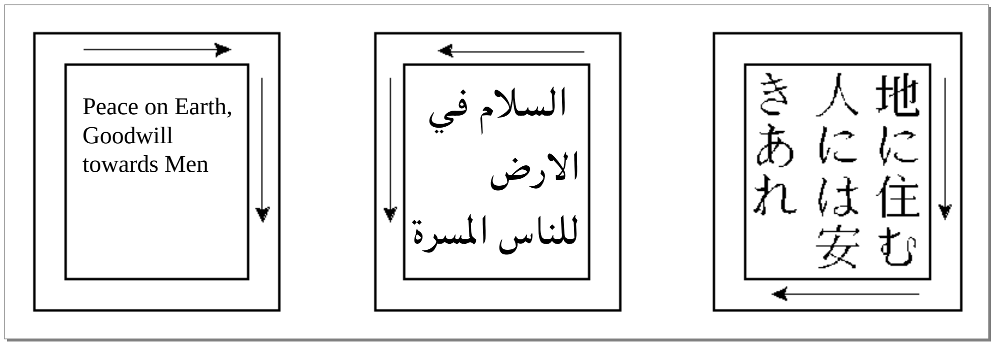
Line Direction and Alignment

Writing systems also vary in the direction in which characters are written:

- In Roman writing systems, characters are written from left to right, with horizontal lines of text filling the page from top to bottom.
- Arabic and Hebrew writing systems have most characters written from right to left, with horizontal lines of text filling the page from top to bottom.
- In Japanese and Chinese, characters are traditionally written from top to bottom, with vertical lines (columns) of characters filling the page from right to left. There are no spaces between words. In modern China and Japan, technical documents and academic journals are written in standard left-to-right horizontal lines, while text for newspapers and magazines is written mostly in vertical columns.
- In Mongolian, the characters are written in a vertical column, with spaces between words, and the lines fill the page from left to right.

Figure 1-14 shows several text directions. These three writing directions—left-right top-bottom, right-left top-bottom, and top-bottom right-left—are the most common of the eight possible combinations of line direction and fill direction.

Figure 1-14 Line directions in text

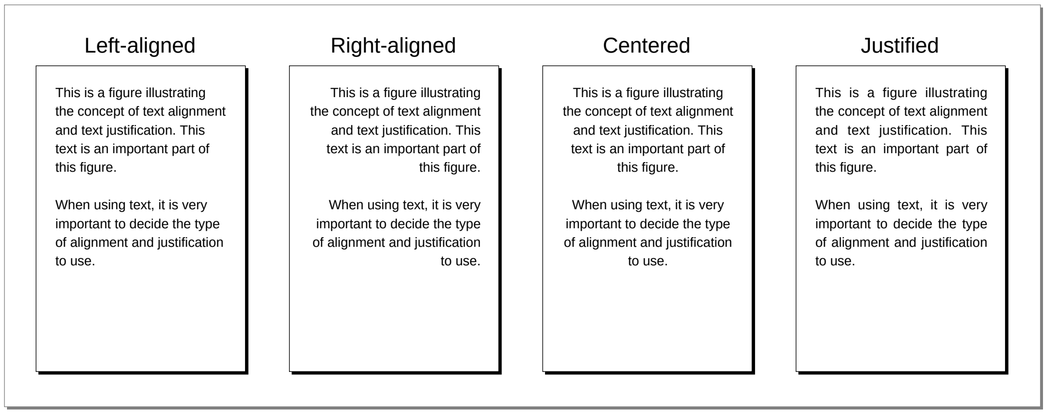


More than one line direction can exist within a single writing system. For example, numbers in Arabic and Hebrew are commonly written left to right, even though nonnumeric text is written from right to left. Furthermore, commonly interspersed foreign words from the Roman writing system are also written from left to right. Thus the Hebrew and Arabic writing systems are actually **bidirectional,** even though their primary line direction is right-to-left.

The Macintosh script management system supports the ability to write text from left to right and from right to left, and to mix text with different directions within lines and blocks of text. Your application can add the ability to handle vertical text, if desired.

Alignment is the horizontal placement of lines of text with respect to the left and right edges of the text area. Alignment can be left-aligned (also called *flush left* or *ragged right*), right-aligned (also called *flush right* or *ragged left*), centered, or **justified** (that is, aligned to both left and right edges of the text area). See Figure 1-15.

Figure 1-15 Text alignment

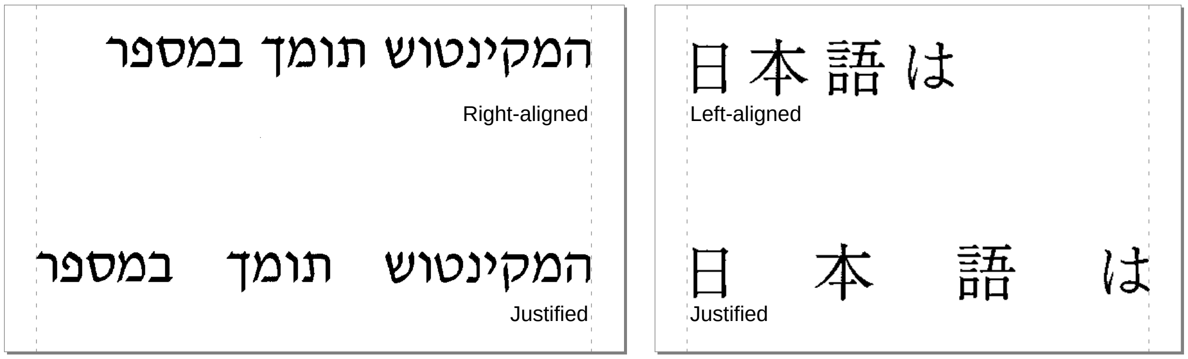

Left-aligned
Right-aligned
Centered
Justified
This is a figure illustrating the concept of text alignment and text justification. This text is an important part of this figure.
When using text, it is very important to decide the type of alignment and justification to use.

Note
Although the term *justified* is sometimes used as a synonym for *aligned*, as in "left-justified" or "right-justified" text, this book considers *justified* to be equivalent only to *fully justified*, and uses *aligned* exclusively when referring to text that is left-aligned, right-aligned, or centered. ◆

Alignment is related to line direction in that text with a left-to-right line direction is usually left-aligned, whereas text with a right-to-left line direction is usually right-aligned.

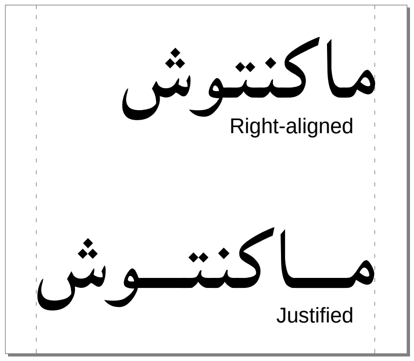
Justification is achieved by spreading or compressing printed text to fit a given line width. It can be performed in Roman text by altering the widths of interword spaces alone, or by altering both interword and intercharacter spaces. Writing systems that don't use interword spaces typically justify text by modifying the intercharacter spacing alone. See Figure 1-16.

Figure 1-16 Justification through interword (Hebrew) and intercharacter (Japanese) spacing



Arabic text, however, is justified by extending characters themselves. Printed or displayed text is justified by inserting extension bar characters (**kashida**) between joined characters, and by widening blank characters to fill any remaining gaps. See Figure 1-17.

Figure 1-17 Justification with Arabic extension bar characters



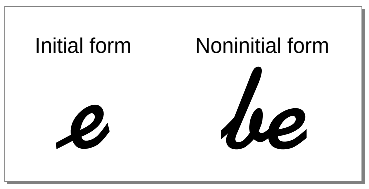
The Macintosh script management system can take all of these justification methods into account when drawing, measuring, or selecting text.

Contextual Forms and Character Reordering

In writing systems, *contextuality* or *context dependence* means that character forms may be modified by the values of preceding and following characters in the input stream. In Arabic, the displayed form of many characters changes depending on their position in a word or on what other characters are nearby.

The displayed form that represents a character in printed English does not usually depend on bordering characters. This is not the case for many writing systems. Even in cursive English, for example, when one letter is joined to the preceding letter, the connecting line varies according to which letters are being joined. Characters may also have considerably different shapes depending on where they occur within a word, for example, at the beginning (initial form) or elsewhere in the word (noninitial form). Figure 1-18 illustrates two of these variations in cursive English, which are called *contextual forms*.

Figure 1-18 Contextual forms in cursive English



The ability to represent contextual forms is required for the proper display of Arabic text. Figure 1-19 shows standalone and contextual forms in Arabic.

Figure 1-19 Standalone and contextual forms in Arabic

Independent	Final	Medial	Initial
ه	ـه	ـهـ	هـ

Furthermore, certain character forms may be combined into a new form when they occur together. Figure 1-20 provides an example of how characters combine to form **ligatures** or *conjunct characters* in Roman text.

Figure 1-20 A ligature in Roman text

$$\mathbf{f} + \mathbf{i} \longrightarrow \mathbf{fi} \longrightarrow \mathbf{ﬁ}$$

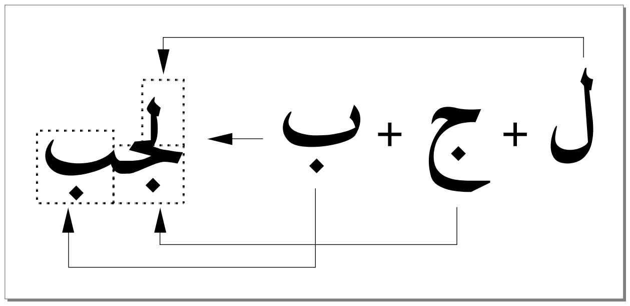
The composition rules for Arabic text, for example, are very complex. The use of ligatures can be highly developed, and some ligatures are required for proper display. Each character can have up to four contextual forms, and the precise form depends upon a varying number of characters that precede and follow it. Figure 1-21 shows an example of a simple ligature in Arabic text.

Figure 1-21 A ligature in Arabic text

لا ← ـا + لـ ← ا + ل

Dozens of Arabic characters form ligatures. As Figure 1-22 illustrates, in some cases, more than two characters can join together into a completely different form, although usually there are only two characters per ligature.

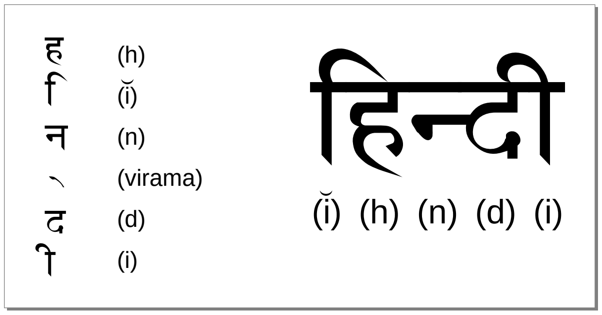
Figure 1-22 A complex ligature in Arabic text



Character reordering is another form of contextuality. Principles of text ordering differ according to the type of writing system under consideration. In most writing systems (including Roman, Greek, Cyrillic, Arabic, and Hebrew), phonetic and writing order are synonymous except for vowel signs and other marks. With certain South Asian writing systems, however, there may be significant differences between phonetic and writing order.

Figure 1-23 shows an example of the reordering of vowels for the word *hindi* in the Devanagari writing system. The left side of the figure shows, in order, the characters that make up the word; the right side shows how the word is actually written. Where there is no explicit vowel sign, consonants take a default vowel sound “a”. To cancel the default vowel, you add a vowel marker (*virama*). Some vowel markers are written to the right of the consonant they modify; others are written to the left, above, or below. In this example, the consonant “h” is followed by a vowel sign, which appears on the *left* when displayed. The consonant “n” is followed by a virama; together they make a small contextual form when displayed. The consonant “d” is followed by a vowel sign, which appears in normal order (on the right).

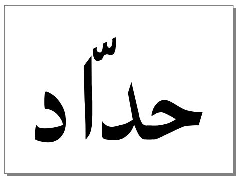
Figure 1-23 Character reordering in the Devanagari writing system



Diacritical Marks

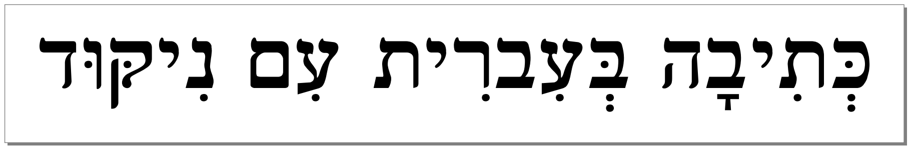
Many writing systems use **diacritical marks,** signs that modify the implicit sound or value of the characters with which they are associated. Some diacritical marks are often referred to as *accents* in Roman writing systems: the acute accent in "é", for instance. Others, such as certain Vietnamese diacritical marks, may indicate pitch, while certain Arabic diacritical marks, such as *shadda* (shown in Figure 1-24), specify extra emphasis on a consonant sound.

Figure 1-24 Arabic text with diacritical mark to specify extra emphasis on a consonant



Hebrew text can contain optional vowel and cantillation marks. Vowel marks are shown in Figure 1-25.

Figure 1-25 Vowel marks in Hebrew text



Uppercase and Lowercase Characters

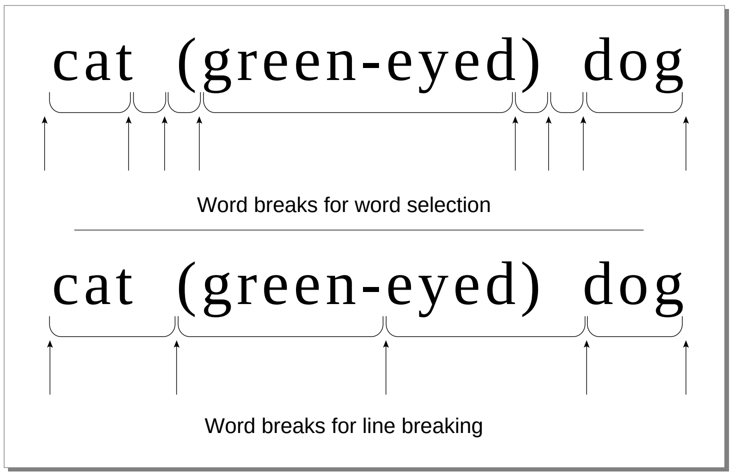
English speakers are familiar with uppercase and lowercase characters in the Roman writing system; however, the majority of the world's writing systems do not have separate uppercase and lowercase forms. The implications for computer applications are primarily in the areas of searching, sorting, and proofreading (for example, spell-checking).

Note
In the Roman writing system, different languages (and even different regions or countries that use the same language) can have different conventions for the treatment of accents and diacritical marks on uppercase characters. These differences are accounted for in individual localized versions of the Roman script system. ◆

Word Demarcation

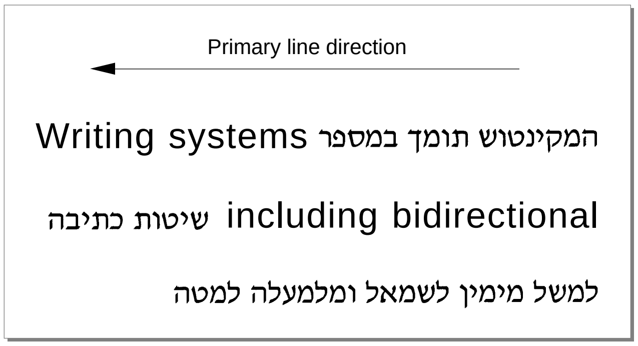
Words in Roman writing systems are generally delimited by spaces and punctuation marks as shown in Figure 1-26. Note also that word demarcation for word selection may follow different rules from word demarcation for line breaking.

Figure 1-26 Word demarcation in the Roman writing system



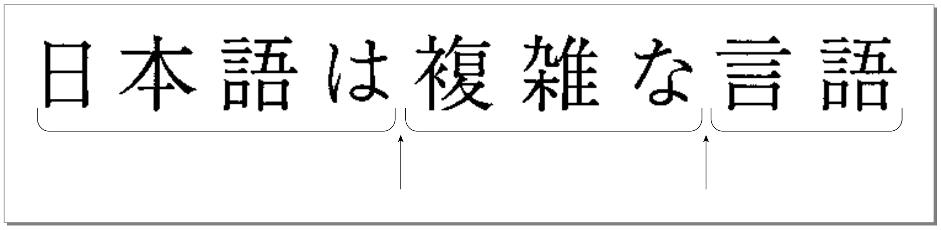
Bidirectional writing systems provide extra challenges to word selection and line breaks. Figure 1-27 shows a single English phrase ("Writing systems including bidirectional") embedded within Hebrew text. The first line breaks within the English text. Note that the line break itself occurs, not at the right or left edge of the first line, but in its interior; and the continuation of the English phrase occurs in the interior of the following line.

Figure 1-27 Line breaking in a bidirectional writing system


Primary line direction
Writing systems המקינטוש תומך במספר
שיטות כתיבה including bidirectional
למשל מימין לשמאל ומלמעלה למטה

In contrast, many Asian writing systems (such as Japanese and Thai) typically have no word delimiters, so the Macintosh script management system provides a more sophisticated method of finding word boundaries. Figure 1-28 shows word demarcation in Japanese.

Figure 1-28 Word demarcation in Japanese



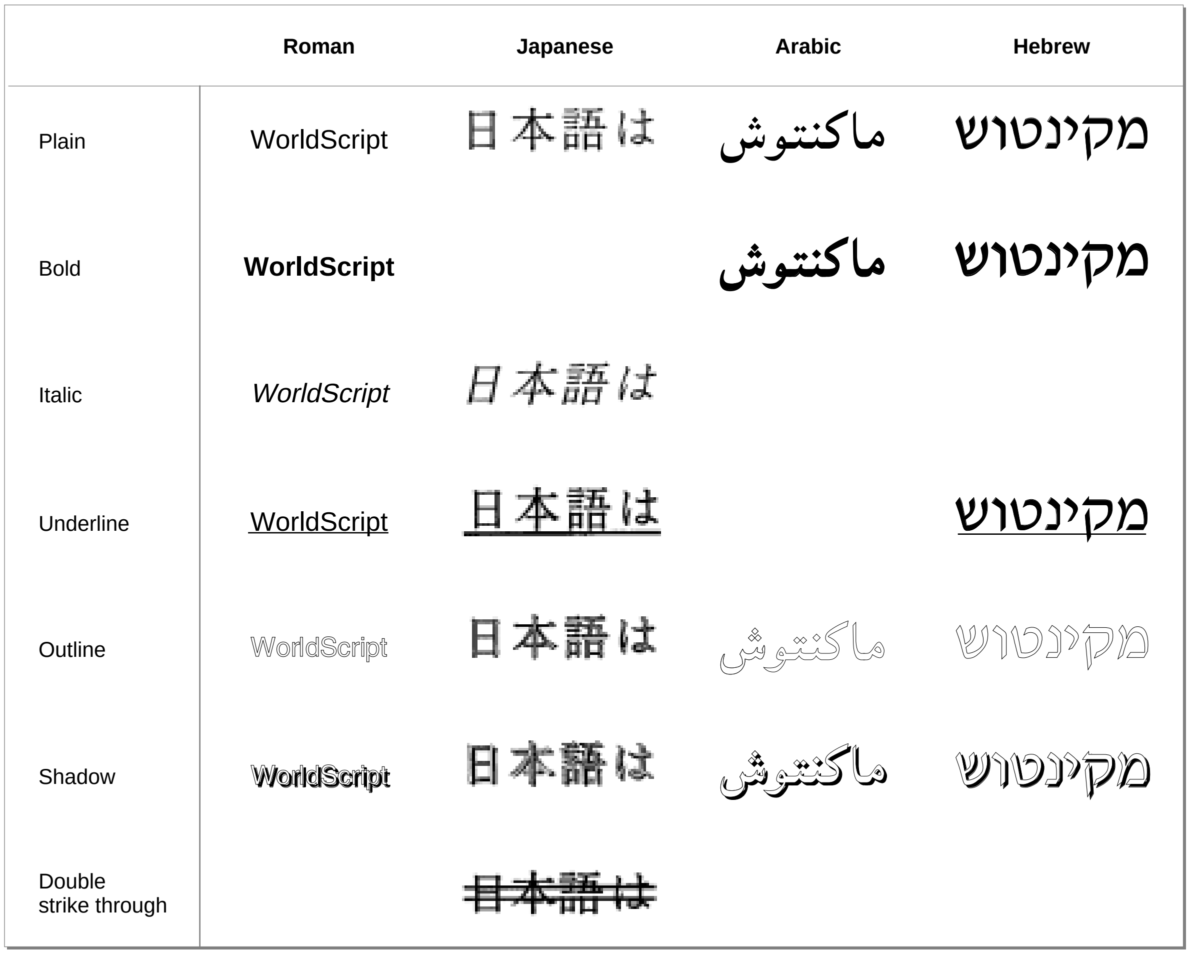
The definition of a word can be an extremely complex issue. Word boundaries are not always well-defined, and native writers of a language may not agree on where particular word boundaries occur.

Styles

Style for a writing system means the systematic alteration of a set of glyphs of a given typeface, to uniformly change their appearance while preserving the overall sense of the typeface. Boldfacing, italicizing, underlining, lining-through, and outlining are possible styles that can be applied to text. Not all styles are appropriate or conventional for all writing systems; for example, underlining may not be meaningful for text that is written in vertical columns, and italicizing may not be appropriate for text that should not be slanted.

Figure 1-29 provides some examples of the application of styles to several writing systems.

Figure 1-29 Selected valid styles in various writing systems



	Roman	Japanese	Arabic	Hebrew
Plain	WorldScript	日本語は	ماكنتوش	מקינטוש
Bold	**WorldScript**		**ماكنتوش**	**מקינטוש**
Italic	*WorldScript*	*日本語は*		
Underline	WorldScript	日本語は		מקינטוש
Outline	WorldScript	日本語は	ماكنتوش	מקינטוש
Shadow	WorldScript	日本語は	ماكنتوش	מקינטוש
Double strike through		~~日本語は~~		

Numbers, Currencies, and Dates

Each language—or in many cases each regional variation of a language—includes a set of conventions for presentation of numbers. For example, in many European countries the decimal character is a comma (,), and the thousands separator is a period (.). In some other areas, western numbers (1…9, 0) are not even used.

Each nation has its own currency format, including the symbol used to denote money. The symbol may be one or more characters, and may precede or follow the numeric amount. Negative monetary values are shown differently in different countries.

Date and time formats vary with language and region. The order in which days, months, and years are written, the words and common abbreviations for days and months, and the separators used in writing dates and times can all differ from region to region.

Figure 1-30 shows some common differences in number, currency, and date formats among the United States, European countries, and Japan.

Figure 1-30 Standard international formats

	Numbers **List separators**	**Currency**	**Time**	**Short date**	**Long date (unabbreviated)** **Long date (abbreviated)**
United States	1,234.56 ;	$0.23 ($0.45) $345.00	9:05 AM 11:20 PM 11:20:09 PM	12/22/85 2/1/85	Wednesday, February 1, 1985 Wed., Feb 1, 1985
Great Britain	1,234.56 ,	£ 0.23 (£ 0.45) £ 345	09:05 23:20 23:20:09	22/12/1985 1/02/1985	Wednesday, February 1, 1985 Wed., Feb 1, 1985
Germany	1.234,56 ;	0,23 DM -0,45 DM 345 DM	09:05 Uhr 23:20 Uhr 23:20:09 Uhr	22.12.1985 1.02.1985	Mittwoch, 1. Februar 1985 Mit, 1. Feb 1985
France	1 234.56 ;	0,23 F -0,45 F 345 F	09:05 23:20 23:20:09	22.12.1985 1.02.1985	Mercredi 1 Février 1985 Mer 1 fev 1985
Greece	1 234.56 ,	*0,23 (0.45) *345	09:05 23:20 23:20:09	22-12-85 1-02-85	Τετάρτη 1 Φερουαρίου 1985 Τετά 1 Φερο 1985
Japan	1 234.56 ;	¥0.23 (¥0.45) ¥345.00	09:05 AM 11:20 PM 11:20:09 PM	85.12.22 85.2.1	1985年2月1日水曜日 1985年2月1日(水)

Even the calendar itself is not the same around the world. The standard **Gregorian calendar** used in Europe and the Americas is not universally accepted:

- In Japan, the Emperor's year is sometimes used instead of the standard Gregorian calendar. The rest of the Japanese calendar system is similar to the Gregorian calendar.
- The Arabic calendar is used extensively throughout the Middle East. It is lunar rather than solar. The months are alternately 29 and 30 days long, so the Arabic calendar year is about 11 days shorter than the Gregorian year. The months have no fixed relation to the sun, so they slowly rotate through all of the seasons of the year (that is, every three years the months shift forward by one Gregorian calendar month).

There are actually two Arabic calendars in common use: the astronomical lunar calendar, based on the moon's phases as actually observed at each location around the world; and the civil lunar calendar, a statutory version of the astronomical calendar. To compute a date correctly for the astronomical lunar calendar requires calculating not only the orbits of the sun and moon, but also knowing the exact latitude, longitude, and time difference from Greenwich mean time.

- Other calendars in common use include the Coptic, Jewish, and Persian calendars.

Character Order and Text Sorting

In most writing systems a need exists for ordering lists of characters, words, or lines of text—such as for writing an alphabet or arranging a dictionary, encyclopedia, or telephone book. Each writing system has its own rules and conventions for sorting text into a meaningful order.

In Roman writing systems, sorting is usually based on alphabetic order, which is fairly simple. However, complications arise when sorting text that includes mixed uppercase and lowercase letters, letters with diacritical marks, ligatures, abbreviations, characters that should be grouped, and characters that should be ignored for sorting purposes.

One important concept for Roman systems is the distinction between primary sorting order and secondary sorting order. Text items that are equivalent in terms of primary sorting characteristics are first grouped, and then differentiated according to secondary sorting characteristics. This allows all variations of a character (uppercase and lowercase, with or without diacritical marks, and so on) to be grouped together in sorted lists.

Nonalphabetic writing systems, such as Chinese or Japanese, can have more complex and less standardized sorting conventions than Roman. Some sorting algorithms for ideographic characters are based on the number of strokes per character. Others are based on *radicals,* standard character subcomponents with a defined sorting order. Others consider the phonetic spelling of the character with Roman or other types of characters (such as Kana), and sort according to Roman alphabetic order or standard Kana order.

Macintosh support for sorting of text is fully described in the chapter "Text Utilities" and the appendixes "International Resources" and "Built-in Script Support" in this book. Tables of specific sorting orders for individual script systems are given in *Guide to Macintosh Software Localization.*

Variations Among Languages and Regions

A writing system by itself may not be enough to define how a language is written. For example, the Roman writing system is used for both the English and French languages. A written **language** refers to the whole body of written words and of methods of combining words, including their meanings, used by a particular group of people. A single writing system may be used by multiple languages. Languages within a writing system can modify the sorting order and word boundaries defined by the writing system, and can define minor modifications to its character set.

Conversely, some languages are written in more than one writing system. The official language of Malaysia, for example, may be written in either the Roman or the Arabic writing system, but the spoken language is called Malay in either case. Romanian and Moldovan are essentially the same spoken language; however, in Romania this language is written in the Roman writing system, whereas in the adjacent republic of Moldova, it is written in the Cyrillic writing system.

A language in itself may not be enough to define all the conventions for written communication in a particular region. A **region** is a linguistic or cultural entity, not necessarily a nation or geographic area, whose written language or other text features are unique enough to be treated separately from other regions. A single language, such as French, may have several regional versions. For example, the French language is used in France, in parts of Belgium, Switzerland, and Canada, and in other countries such as Luxembourg, Haiti, Mali, Zaïre, Tahiti, and Vanuatu. Such different areas that use the same language may have different conventions for time, date, and number formats, as well as rules for case conversion or placement of diacritical marks. Some differences may also occur in the behavior of the written language. For example, in France, accents on most characters are generally omitted if the character is written in uppercase; in Québec, the accents are usually preserved.

The Macintosh script management system can account for multiple languages and regional variations within script systems. See “Script Codes, Language Codes, and Region Codes” beginning on page 1-48.

Components of the Macintosh Script Management System

This section describes the organization of the Macintosh script management system, those parts of the system software that provide support for the writing-system features described in the previous section, “Features of the World’s Writing Systems.”

The Macintosh script management system makes it possible to represent many writing systems and languages on the Macintosh computer. With the Macintosh script management system, your application’s text-manipulation capabilities can extend far beyond the Roman writing system and its languages. If you use its features your application can have a much wider market worldwide. You can implement text-handling capabilities that work properly with any supported writing system, or you can tailor your application to work correctly with any specific writing system or any regional variation of a writing system.

The script management system supplies much of the same basic capability for entering and displaying text as does a multi-language word processor—but on a system level. Since the capability is built into the system, you do not have to duplicate the code necessary to support each writing system; instead, you can devote your efforts to the primary functions of your application.

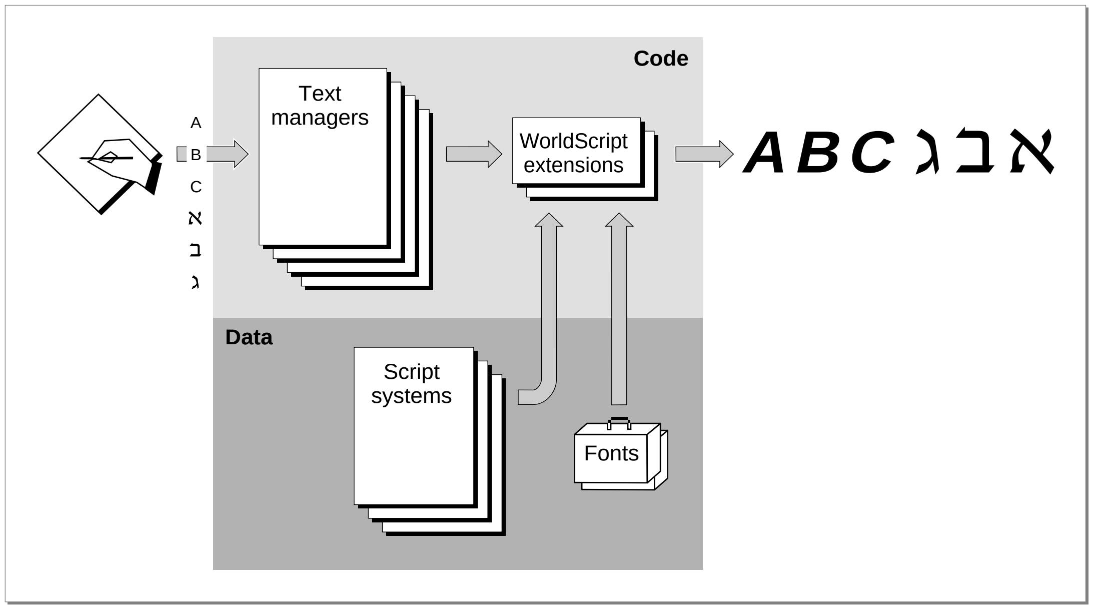
As Figure 1-31 shows, the script management system consists of

- routines in various components (managers) of system software
- two WorldScript extensions (optional)

- one or more script systems
- one or more fonts

The text managers and the script extensions are mostly code; they execute the script-aware calls your application makes when handling text. The script systems and fonts are mostly data; they consist largely of tables of script-specific information used by the text routines, and glyph descriptions.

Figure 1-31 Components of the script management system for text display



The Macintosh Text Managers

Several parts of Macintosh system software work together to provide specific text-handling services to your application. These text-related managers include the Script Manager, the text-handling components of QuickDraw, the Font Manager, the Text Utilities, the Text Services Manager, and the Dictionary Manager.

The Script Manager

The Script Manager is at the center of the Macintosh script management system. It initializes script systems and makes them available to applications; it maintains important data structures and provides a standard application interface to script systems; it supports switching text input among different script systems; and it provides several text-manipulation services.

The Script Manager works closely with the Text Utilities and with QuickDraw. Your program typically makes calls to all three managers in the course of text-handling, and in many cases a call to one of these managers results in internal calls among them. TextEdit also relies on the Script Manager, Text Utilities, and QuickDraw to make sure that it handles text correctly in any script system.

The Script Manager provides routines with which you can

- control the values of many script-related settings, including the *system direction* and the keyboard script
- get information about script systems, such as script codes, character-type information, and direct access to a script's international resources
- modify text through lexical conversion to tokens or phonetic conversion within a script system
- modify script systems by replacing international resources or routines

In particular, the Script Manager gives you access to *Script Manager variables,* which control many overall settings of the text environment, and *script variables,* which control settings specific to each enabled script system.

QuickDraw

QuickDraw is the graphics manager of Macintosh system software. The graphics components of QuickDraw are described in the QuickDraw chapters in *Inside Macintosh: Imaging;* the text-handling components of QuickDraw are described in the chapter "QuickDraw Text" in this book.

Your application makes QuickDraw calls to write text to the screen or to a printer. When QuickDraw draws text, it places bitmapped shapes on the display device that represent the characters it is drawing. The characters are drawn according to the settings of the currrent window's graphics port record, which includes the location at which to draw and a specification of the font and character attributes with which to draw.

For text in various script systems, the QuickDraw text routines allow you to

- set the characteristics of the drawing environment
- draw text
- measure the width of text
- lay out lines of text
- determine caret positions and highlight text

Font Manager

QuickDraw cannot draw text without a font. The Font Manager supports QuickDraw by providing the character bitmaps that QuickDraw needs, in the typefaces, sizes, and styles that QuickDraw requests. The Font Manager keeps track of all fonts available to an application. The Font Manager supports fonts in many languages, for both bitmapped and outline fonts, and for both 1-byte and 2-byte fonts.

Besides providing QuickDraw with the bitmaps it needs, the Font Manager provides routines with which you can

- determine the characteristics of a font
- change certain font settings, such as fractional widths or scaling
- favor outline fonts over bitmapped fonts
- manipulate fonts in memory

Text Utilities

The Text Utilities are an integrated collection of routines for performing a variety of operations on text, ranging from sorting strings to formatting dates and times to finding word boundaries. The Text Utilities work in conjunction with the Macintosh script management system and can take into account the differences in text-handling among script systems. If you use these routines you can handle text operations in a manner that is transportable to different parts of the world.

Many of the Text Utilities routines are script-aware; they work in conjunction with the Script Manager and with QuickDraw to determine the script-system characteristics of text and to prepare the text for drawing to the screen or printing.

The Text Utilities provide routines that, for text in any script system, allow you to

- define strings in various ways
- compare and sort strings
- modify the contents of strings by truncation, stripping of diacritical marks, case conversion, or replacement
- find boundaries of words, lines, and runs of Roman characters
- convert and format date and time strings
- convert and format numeric strings

Text Services Manager

The Text Services Manager is the part of Macintosh system software that provides an environment for applications to use text services such as input methods. The Text Services Manager handles communication between client applications and text service components. **Text service components** are specialized software modules for entry, processing, or formatting of text.

Client applications can use the Text Services Manager to

- make text services available to the user
- search for and communicate with text service components
- accept text input or other information from text service components
- ask for a special floating input window service

Text service components can use the Text Services Manager to

- make their text service available to an application
- act on events involving their windows, menus, or cursor
- pass text input or other information to an application
- display floating utility windows

Dictionary Manager

The Dictionary Manager is the part of Macintosh system software that allows you to create dictionaries for input methods and other text services that let the user enter, format, and process text. A dictionary is a data file with information essential to the conversion of text from one form to another. Most input methods provide both a **main dictionary,** which contains standard information for conversion between forms, and a **user dictionary,** which allows users to add custom information.

The Dictionary Manager defines a uniform and public dictionary format that you can apply to your text service's dictionaries. The Dictionary Manager provides routines with which you can

- create and access a dictionary
- locate, insert, or delete records in a dictionary
- compact data in a dictionary

The WorldScript Extensions

The Roman script system, always available on every Macintosh, needs only the previously mentioned managers to function correctly. Several other similar script systems also need no other software. However, for those 1-byte script systems that have contextual characters or right-to-left line direction, additional code is needed so that the Script Manager, Text Utilities, and QuickDraw routines can work properly. Likewise, 2-byte script systems need code extensions in order to properly handle the thousands of characters they use.

Although each writing system has unique requirements and procedures for presenting, sorting, and formatting text, in many cases separate script systems can use similar algorithms. Therefore, to avoid inconsistencies and unnecessary duplication of code, the script management system supplies two system extension files—WorldScript I and WorldScript II—that support 1-byte complex script systems and 2-byte script systems, respectively (see "Types of Script Systems" on page 1-46). They contain code that implements many script-aware text-manipulation routines, eliminating the need for each script system to maintain its own code extensions. Script-specific behavior is encoded in resource-based tables accessed by the extensions.

WorldScript I and WorldScript II are described in the appendix "Built-in Script Support" in this book.

WorldScript I

WorldScript I is the script extension that implements table-driven text measuring and drawing behavior for all 1-byte complex script systems (such as Hebrew, Arabic, Thai, and Devanagari). Using tables in each script system's international resources, WorldScript I performs text manipulation properly for all supported scripts. WorldScript I is a single file located in the Extensions folder within the System Folder on the user's Macintosh. It installs all compatible 1-byte script systems that are present in the System file, and provides them with a standard set of script-aware text-manipulation routines.

WorldScript I implements **script utilities,** the low-level routines through which an individual script system implements script-aware Text Utilities, QuickDraw, and Script Manager routines. WorldScript I also implements patches to certain QuickDraw and Font Manager text-handling routines.

The Script Manager provides routines that allow you to modify or replace a 1-byte complex script system's script utilities and QuickDraw patches. See the chapter "Script Manager" in this book.

WorldScript II

WorldScript II is the script extension that implements table-driven text measuring and drawing behavior for all 2-byte (Chinese, Japanese, Korean) script systems. Using tables in each script system's international resources, WorldScript II performs text manipulation properly for all supported scripts. WorldScript II is a single file located in the Extensions folder within the System Folder on the user's Macintosh. It installs all compatible 2-byte script systems that are present in the System file, and provides them with a standard set of script-aware text-manipulation routines.

Like WorldScript I, WorldScript II implements script utilities that implement script-aware Text Utilities, QuickDraw, and Script Manager routines. Unlike WorldScript I, WorldScript II does not support the Script Manager routines that allow replacement of script utilities.

Components of a Script System

The Macintosh script management system, as described in the previous section, is designed to manipulate text according to information contained in script systems. This section describes how script systems are organized.

A Macintosh script system is a collection of resources, mostly tables of data, that defines the behavior of a particular writing system. The script system specifies the character set, sorting orders, date and number formats, line direction, character reordering, accent placement, and other writing-system-specific features. Your application uses the information in a script system when it makes a script-aware text-handling call, and it can also access the resources of a script system directly, to inspect or modify its behavior.

Each Macintosh script system consists of a set of international resources and a set of keyboard resources. In addition, a script system requires one or more fonts in order to display its text. A script system may also have a control panel device through which the user can configure the individual characteristics of the script at any time.

Resources in general are described in the chapter "Resource Manager," in *Inside Macintosh: More Macintosh Toolbox.*

International Resources

The international resources are a set of Macintosh resources that specify text handling and display information for a particular writing system, language, or region. Such information includes number and currency formats, long and short date formats, preferred sorting order, character type, case conversion, and word-boundary information.

Table 1-1 lists the international resources, shows their resource types, and summarizes their contents.

Table 1-1 The international resources

Name	Resource type	Content
International configuration	`'itlc'`	Configuration of the system script, plus Script Manager flags, and the region code for the system script
Script sorting	`'itlm'`	Tables showing sorting order and mapping among script systems, languages, and regions
International bundle	`'itlb'`	IDs of all required resources for a script system, plus bit flags, default language, and other settings
Numeric format	`'itl0'`	Number and currency formats, short date and time formats, unit of measurement for a script system, plus a region code
Long-date format	`'itl1'`	Long date and time formats, names of days and months for a script system, plus a region code
String manipulation	`'itl2'`	Sorting routines, tables for character type, case conversion, and word boundaries for a script system
Tokens	`'itl4'`	Tables and code for converting characters to tokens and back in a script system, and for formatting numbers
Encoding/rendering	`'itl5'`	Tables for character rendering (for 1-byte script systems); tables for character encoding (for 2-byte script systems)
Transliteration	`'trsl'`	Tables for phonetic conversion among subscripts of a 2-byte script system

International resources reside in the resource fork of the Macintosh System file. However, not every installed script system requires a complete set of them:

- There is only one international configuration resource for each Macintosh System file, and its resource ID is 0. It configures the system and defines the system script.
- There is only one script-sorting resource for each Macintosh System file, and its resource ID is 0. It does not belong to any script system.
- Each installed script has one international bundle resource. Its resource ID is the script code of the script system it implements.
- Each installed script system has one or more numeric-format, long-date-format, string-manipulation, and tokens resources. Their resource IDs are in a range that defines the script system they belong to; see Figure 1-35 on page 1-50.
- A script system may have one or more optional encoding/rendering and transliteration resources.Their resource IDs are also in a range that defines the script system they belong to.

A single script system may have multiple localized versions of its `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, `'itl5'`, and `'trsl'` resources, in order to represent different languages or regional variations of the script. You can manipulate text in different formats within that script system by switching among the multiple versions of the resources. See “Installing Modifications to a Script System” beginning on page 1-103.

See the appendix “International Resources” in this book for more information on international resources.

Keyboard Resources

The keyboard resources are a set of Macintosh resources that specify how keyboard input is converted to text for a particular writing system, language, or region. The Event Manager, the Script Manager, and the Menu Manager use the information in these resources to convert keypresses to character codes, to switch input among different script systems, and to display the icon of the current keyboard in the Keyboard menu. The Resource Manager, the Event Manager and the Menu Manager are described in *Inside Macintosh: Macintosh Toolbox Essentials*; the Resource Manager is described in *Inside Macintosh: More Macintosh Toolbox*.

Table 1-2 lists the keyboard resources, shows their resource types, and summarizes their purpose.

Table 1-2 The keyboard resources

Name	Resource type	Content
Key map	`'KMAP'`	Maps hardware-dependent raw key codes to hardware-independent virtual key codes
Key remap	`'itlk'`	Remaps some virtual key codes from certain keyboards for use by some keyboard-layout resources
Keyboard layout	`'KCHR'`	Maps virtual key codes to character codes; represents the character set for a script system
Keyboard icons	`'kcs#'`	Keyboard icon list for black-and-white icon display in the Keyboard menu
	`'kcs4'`	Keyboard icon list for 4-bit color/gray-scale icon display in the Keyboard menu
	`'kcs8'`	Keyboard icon list for 8-bit color/gray-scale icon display in the Keyboard menu
Keyboard swap	`'KSWP'`	Specifies modifier-plus-key combinations to let the user change keyboard layout, keyboard script, or input method
Key caps	`'KCAP'`	Determines keyboard display for a given physical keyboard (in Key Caps desk accessory)

Keyboard resources reside in the resource fork of the Macintosh System file. Some are script-related, but others are hardware-related and script-independent:

- There is only one keyboard-swap resource per Macintosh System file. Because it specifies how to switch among script systems, it does not belong to any script system. Its resource ID is 0.
- There is one key-map resource that supports most types of physical keyboards. Some keyboards need their own key-map resource, in which case the key-map resource ID is equal to the ID number of the keyboard it is associated with.
- There is one key-caps resource for each type of physical keyboard available. It is independent of any script system. Its resource ID is equal to the ID number of the keyboard it is associated with.

- There are one or more keyboard-layout resources per script system. There are one or more families of keyboard icon resources per script system (one per keyboard layout resource or input method). The resource ID for each keyboard-layout resource is in a range that defines the script system it belongs to; see Figure 1-35 on page 1-50. The resource ID for each keyboard icon family is equal to the ID of its associated keyboard-layout resource.
- There is one key-remap resource for each keyboard-layout resource that needs one. Its resource ID is equal to the ID number of its associated keyboard-layout resource.

See the appendix “Keyboard Resources” in this book for more information.

Fonts

A font is not technically part of a script system. The script’s international bundle resource does not have to specify any particular font resource IDs, and even if it does, their presence is not guaranteed. Nevertheless, no script system can be used unless one or more fonts accompany it.

A Macintosh font implements the character set and other written forms such as ligatures for a given script system. Each font contains a particular set of glyphs that share certain design characteristics. Those glyphs constitute a typeface, and the typeface has a name, such as Times, Helvetica®, or Kyoto. A font may be a plain implementation of a typeface, or it may be styled—such as bold or italic. (QuickDraw can also *produce* styled versions of the characters of a typeface from a plain font.)

Glyphs in a font represent each of the characters of a character set. Additional glyphs may be present to represent ligatures, and other contextual forms. In some fonts there may be more contextual glyphs than character glyphs.

A font maps character codes to glyphs, and may contain tables that map special glyph codes to the glyphs of contextual forms. When laying out and drawing text, the script management system uses information in the font to convert character codes in memory to a properly formatted series of glyphs on the screen or on the page.

Macintosh fonts are either bitmapped (meaning that each glyph is a single bitmap) or outline (meaning that each glyph is a mathematical outline that is size-independent). A bitmapped font contains a single set of glyphs at a fixed size, whereas one outline font can produce glyphs of any size.

Fonts are either 1-byte—meaning that they have glyphs for 256 or fewer characters—or 2-byte, meaning that they can have glyphs for thousands of characters. The 1-byte fonts represent character codes that are 1 byte long, and include all fonts of the Roman script system. The 2-byte fonts represent character codes that are 1 byte or 2 bytes long, and include fonts of the Chinese, Japanese, and Korean script systems. The script management system supports 2-byte fonts, and can correctly handle mixtures of 1-byte and 2-byte characters in text.

Each font is a Macintosh resource. For ease of reference, fonts are grouped into **font families** (resource type `'FOND'`). Each family consists of all the available sizes and styled variations of a single named typeface. For example, "Courier 10", "Courier 12 Italic", and "Courier Semibold" could be two bitmapped fonts and one outline font belonging to the single font family "Courier". Whenever you supply a font ID to a script management call, it is the `'FOND'` resource ID that you supply (unless you supply the special font designators 0 or 1; see page 1-61).

As with other script-related resources, the ID numbers for fonts are in a range that defines the script system they belong to. See, for example, Figure 1-35 on page 1-50. In fact, the script management system relies fundamentally on font family ID to determine the script system associated with any text that is to be manipulated or drawn.

See the section "Font Handling" beginning on page 1-60 of this chapter for more information and programming suggestions involving fonts. See the chapter "Font Manager" for more specific information on font structure and use. For more complete information on both 1-byte and 2-byte TrueType fonts, see *The TrueType Font Format Specification,* available from APDA.

How Script Systems Are Classified

Different kinds of script systems function differently. The previous section, "Components of a Script System," described the components of all script systems; this section describes the different ways of classifying script systems. The following section, "How Script Systems Work," describes how to use script systems for your text handling needs.

Script systems are typed in general by the size of their character set and by their relative similarity to Roman. The Roman script system is used widely in North America, South America, Australia, Europe, and Africa, and in parts of Asia and Oceania. The Roman script system is standard on all Macintosh system software versions 4.1 and higher, but the Macintosh also supports all types of non-Roman script systems—simple or complex, and with small or large character sets.

Script systems are individually classified by code numbers. Resources associated with a script system have ID numbers that are related to the script's code. Languages and regional variations are subsets of script systems and have their own code numbers.

On an individual computer, more than one script system can be available at a time; different scripts are classified by their function. The most important script system is the *system script;* other script systems are secondary. The script system currently being used for text display is the *font script;* the script currently being used for text input is the *keyboard script.*

Note
Because the Roman script system is always installed, you can always manipulate Roman text, no matter what other script systems are present. ◆

Types of Script Systems

Because of its historical support for the Roman writing system, and because Roman text layout is fairly simple, the Macintosh computer most easily supports script systems that are like Roman. Other script systems can add complications like right-to-left line direction, contextual character forms, and large character sets.

As shown in Figure 1-32, script systems are divided into three groups, based on the size of their character set and their relative complexity compared to Roman:

- The **1-byte simple script systems** have character sets of 256 characters or fewer. They are called 1-byte because their character codes are one byte long. They are called simple because they are similar to Roman: they have a uniform left-to-right line direction and are noncontextual. The 1-byte simple script systems support variations within the Roman writing system and among Roman-like writing systems such as differences of character set, keyboard layout, sorting order, word boundaries, and the formatting of dates, times, and numbers. The 1-byte simple script systems include Roman, Greek, and Cyrillic.
- The **1-byte complex script systems** also have character sets of 256 characters or fewer. They are complex because they may have left-to-right or right-to-left line direction, and may be contextual. The 1-byte complex script systems support the more difficult formatting required for bidirectional writing and the extensive use of ligatures, cursive fonts, character reordering, and other contextual features. The 1-byte complex script systems include Thai, Devanagari, Hebrew, and Arabic.
- The **2-byte script systems** have character sets so large that most character codes are two bytes long. The 2-byte script systems require sophisticated methods for character input, as well as an independent font mechanism for display and printing. The 2-byte script systems include Traditional Chinese, Simplified Chinese, Japanese, and Korean.

Figure 1-32 Types of script systems

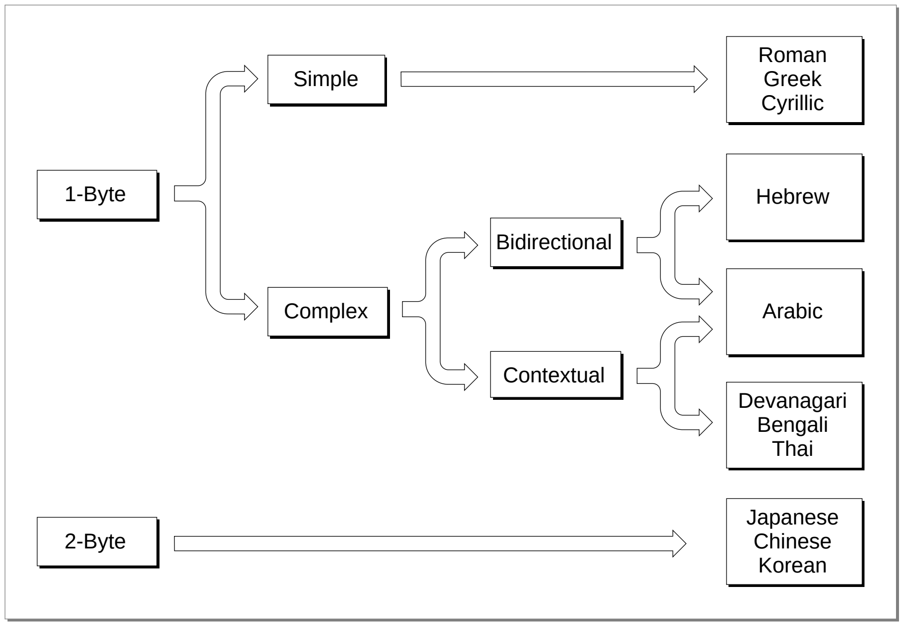
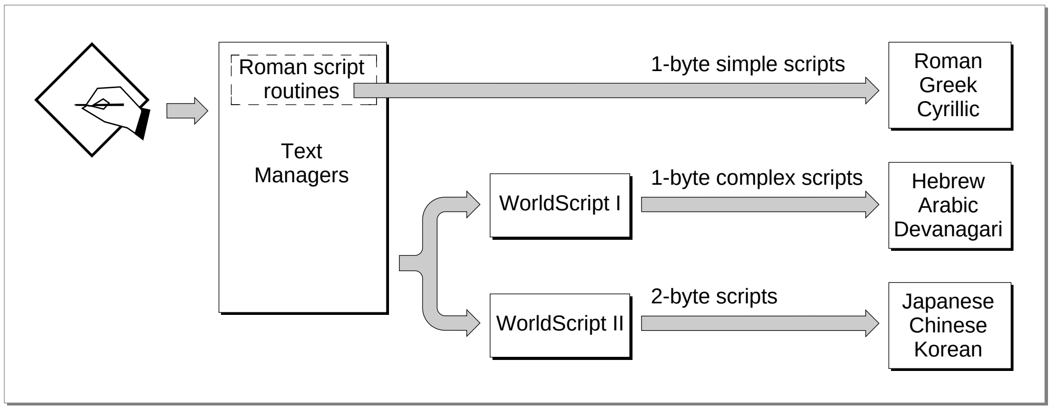


Figure 1-33 shows which parts of the Macintosh script management system are involved in handling text from the different types of script systems:

- Roman text is handled with code and resources largely built into system software.
- Text of 1-byte simple script systems is handled with the same built-in Roman code and resources, supplemented by minor additional resources such as alternate keyboard layouts and fonts.
- Text of 1-byte complex script systems is handled by WorldScript I, which may use, modify, or completely replace any of the built-in code. The complex 1-byte script systems may replace much of the Roman resources with their own international and keyboard resources and fonts.

- Text of 2-byte script systems is handled by WorldScript II, which may use, modify, or completely replace any of the built-in code. The 2-byte script systems may replace much of the Roman resources with their own international and keyboard resources and fonts; they also provide special input methods for text entry.

Figure 1-33 How the script management system handles different types of scripts



Script Codes, Language Codes, and Region Codes

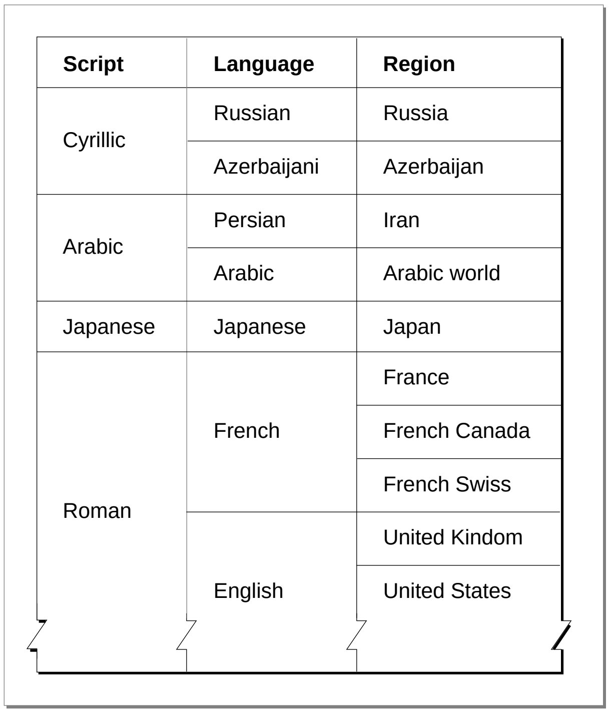
The Macintosh script management system accommodates the international differences within writing systems by defining languages and regional variations for script systems, and organizing them into a classified hierarchy. Script systems are identified by **script codes,** languages by **language codes,** and regions by **region codes.** A spoken language that may be written in more than one writing system is treated on the Macintosh computer as several languages, each belonging to a different script system.

Three general concepts underlie the hierarchy of script, language, and region.

- A script system is often differentiated by its character encoding, the specification of the characters that compose the writing system and their numeric representations. Different character encodings usually have different script codes. (This is not always true within the Roman script system; see “The Standard Roman Character Set” on page 1-54.)
- Each language belongs to a particular script system. Every language code thus implies a particular script code. Several languages may be associated with a single script system; in such a case, they share the same character set.
- A region code designates an area that may be smaller or larger than a single country (for example, *French Swiss* or *Arabic*), in which a specific variation of a single script system and language is used. Each region belongs to a particular language. Several regions may be associated with a single language. A region code typically represents a localized version of the system software for a particular language in a particular country or region.

Figure 1-34 illustrates the script, language, and region hierarchy. Note, for example, that the regions of France, Québec, and French Swiss are associated with the French language, which is part of the Roman script system.

Figure 1-34 The script, language, and region hierarchy



Script	Language	Region
Cyrillic	Russian	Russia
	Azerbaijani	Azerbaijan
Arabic	Persian	Iran
	Arabic	Arabic world
Japanese	Japanese	Japan
Roman	French	France
		French Canada
		French Swiss
	English	United Kindom
		United States

You can use language codes and region codes to specify multiple subsets of the international resources for a single script system. That way you can implement regional variations to a writing system without having to create an entirely new script system each time. See "Installing Modifications to a Script System" beginning on page 1-103.

See the chapter "Script Manager" in this book for a complete list of the constant names that define the codes for all scripts, languages, and regional versions.

Script Codes and Resource ID Numbers

Each script system is assigned a unique script code. The script codes currently defined are in the range 0–32, although the Script Manager can support 64 script systems at the same time. All the resources related to a script system, including its fonts, have resource ID numbers related in some way to the script ID:

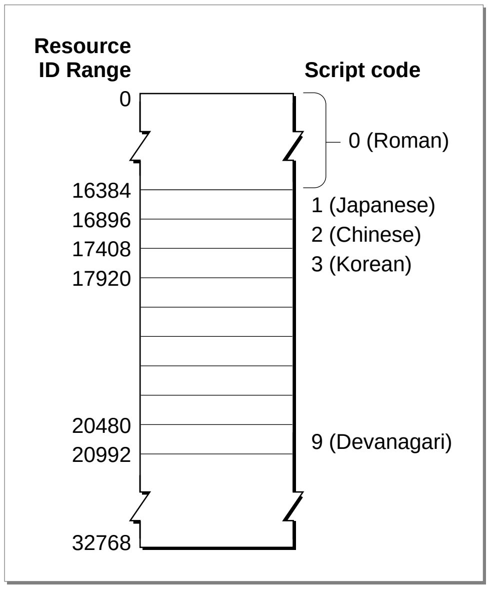
- The resource ID number for a script system's international bundle (`'itlb'`) resource is the same as the script code.

- The resource ID numbers for most other resources associated with a script are in a range specific to that script. You can use these ID ranges to determine the script system associated with a font or other resource. Likewise, even when a font is missing, the Font Manager can use the ID range to substitute a font of the same script.
  - For Roman (script code = 0), this range is 0–16383.
  - Scripts with script codes in the range 1–32 have a range of 512 resource ID numbers each. For example, the script code for Japanese is 1, so Japanese resources can have any of the first 512 ID numbers beyond the Roman range, that is, 16384–16895. The script code for Korean is 3, so Korean resources can have resource IDs in the range 17408–17919.

Figure 1-35 illustrates the resource ID ranges for script systems with script codes between 0 and 32. The ranges for the Roman, Japanese, Chinese, Korean, and Devanagari scripts are noted. A full table of resource ID ranges is provided in the appendix “International Resources” in this book.

Note
The Script Manager provides routines for determining the script system based on the value of a font family ID. ◆

Figure 1-35 Distinguishing scripts by resource ID range (for script codes 0–32)



IMPORTANT
The special font designators *0* and *1*, although in the range of the Roman script system, specify the Macintosh *system font* and *application font*, respectively; they do *not* necessarily indicate a Roman font and the Roman script system. See the section “Font Handling” beginning on page 1-60 for more information. ▲

The System Script and Auxiliary Scripts

A script system may be installed either as an **auxiliary script** (also called a *secondary script*), which only provides support for a particular writing system, or as the **system script** (also called the *primary script*), which is the script system associated with the currently running version of Macintosh system software. The system script affects system defaults such as the default font, keyboard layout, and primary line direction. The system script defines which writing system is used for dialog boxes, menus, and alerts. Therefore, most text displayed by the Finder and other parts of the system is in the language of the system script.

The system script is specified in the System file's international configuration (`'itlc'`) resource. All other script systems are secondary to the system script. In non-Roman versions of system software, Roman is an auxiliary script.

Some versions of Macintosh system software, such as the Turkish or French, are simply variations of the U.S. system software (which includes the Roman script system). Their script system is a modified version of the standard U.S. Roman script system, and they do not include a second script system. When a non-Roman script system is installed, however, at least two script systems are always present. For example, the Japanese system software is a combination of U.S. system software and the Japanese script system, all of which are localized for Japan. Thus it contains both Roman and Japanese script systems.

Font Script and Keyboard Script

In every version of Macintosh system software, the system script is always enabled and is the principal script system for determining how text is presented and handled in the Finder and other parts of system software. But if there are auxiliary scripts present, the system script is not always the script system that controls text-handling.

The text-manipulation and drawing routines in the Macintosh script management system work with individual character codes or strings of character codes, manipulating them or converting them to glyphs. A character code by itself carries no identifier as to what script system should be used to interpret it; the script management system uses other information to decide what script system to use for presenting or processing a given run of text.

Many of the routines use the script system associated with the font of the current graphics port to perform their tasks. The font is specified by the `txFont` field of the graphics port that is identified by the global variable `thePort`. The script system associated with that font is called the **font script.** Therefore, to manipulate text in a given script system, you typically first set the current port with a call to the QuickDraw `SetPort` procedure, and then set the current font with a call to the QuickDraw `TextFont` procedure, and then call the series of text-manipulation routines you need. (For those routines that take a script code as an explicit parameter, you need not set the current font before making the call.)

Text input by the user involves the conversion of keypresses to character codes. Because every script system has its own character set, the character codes produced depend explicitly on the script system used for keyboard input. That script system is called the **keyboard script.** It is not automatically the same as the script used for display of text; your application must keep the keyboard script and the font script synchronized if characters are to be displayed correctly as they are typed in. Synchronization of the font script and keyboard script is further described on page 1-90 in this chapter and in the chapter "Script Manager" in this book.

What is the "current" script?

As just stated, the font script is usually the script system that is used by a script-aware text routine when the identity of the script or its resources is not an explicit parameter of the call. However, if the font script is not enabled, the routine uses the system script by default. Furthermore, some script-aware routines may use the system script instead of the font script, depending on the values of two Script Manager flags: the **font force flag** and the **international resources selection flag.**

The font force flag, when `TRUE`, specifies that fonts with ID numbers in the Roman range are to be considered as fonts of the system script rather than Roman fonts. The international resources selection flag, when `TRUE`, specifies that resources of the system script are to be used by those Text Utilities routines that format dates, times, and numbers. The font force flag is supported only by some non-Roman 1-byte scripts for special purposes, and is typically `FALSE`. The international resources selection flag is typically `TRUE`.

The font force flag and the international resources selection flag are described in the chapter "Script Manager" in this book. ◆

How Script Systems Work

The previous sections, "Components of a Script System" and "How Script Systems Are Classified," described the organizational aspects of script systems. This section explains how Macintosh script systems function in support of the world's writing systems. It discusses how script systems represent the multitude of characters in the world's languages, how they format and draw those characters in the context of surrounding text, how they support user input of text, and how they handle text-manipulation such as sorting and searching across many languages.

Character Encoding

Character encoding is the organization of the set of numeric codes that represent all the meaningful characters of a script system in memory. Each character is stored in memory as a number. When a user enters characters, the user's keypresses are converted to character codes; when the characters are displayed onscreen, the character codes are converted to the glyphs of a font.

There are two fundamental classes of character encodings supported by Macintosh system software: 1-byte and 2-byte. A 1-byte encoding represents every character with a 1-byte number; a 2-byte encoding (actually a mixed encoding) represents characters with either 1-byte or 2-byte numbers. There can be up to 256 characters in a character set that has 1-byte encoding, whereas there can be over 28,000 characters in a character set that has the currently supported 2-byte encoding. Roman and many other script systems use 1-byte encodings; Chinese, Korean, and Japanese script systems use 2-byte encodings.

The meaning of each character code is unique only within its script system. In an Arabic font, the code $CC represents the character *jiim*, and in a standard Roman font, the code $CC represents the character Ã. The traditional Chinese and simplified Chinese script systems are two different script systems and use different character encodings; the Chinese characters used in the Japanese and Korean script systems have still different character encodings.

Much of a script system's behavior, including sorting and composition rules for drawing and measuring, is encoded in tables that rely on a particular order of character codes. Therefore, the character encoding is fixed; it cannot be changed without significant consequences. Ideally, each script system is consistent in its character encoding; all fonts within a script system should have identical font layouts that reflect that encoding. This is largely true, with the exception of some Roman fonts; Symbol font, for example, is a Roman font but its glyphs are completely different from those of other Roman fonts.

The character set of a script system can include the characters of one or more subscripts. A **subscript** is a portion of a script system that has its own character set and conventions for use. Subscripts within the Japanese script system, for example, include the **Katakana** and **Hiragana** syllabic characters. All non-Roman script systems include Roman as a subscript. The parts of a script system's character set that implement its natural writing system are called **native** characters. In the Arabic script system, Arabic characters are native and Roman characters constitute a subscript.

The Unicode standard

Unicode is an ISO standard for 16-bit universal worldwide character encoding. It has been developed by a consortium that includes Apple Computer, Inc. In the future, Unicode will replace individual script systems' character encodings with one complete 16-bit character encoding applicable worldwide to all characters in all languages. The script systems described in this book do not yet use Unicode encodings.

With a universal character encoding such as Unicode, the character sets of separate writing systems do not overlap; there is no need to define script systems, because each character code by itself determines which writing system the character is part of. Furthermore, Unicode takes care of the problem of conflicting character encodings within a single writing system; for example, in Unicode, there is no overlap between Roman character codes and the codes of the symbols in Symbol font. ◆

The Standard Roman Character Set

The Apple **Standard Roman character set** is the 1-byte character encoding for the Roman script system. It is the fundamental character set for the Macintosh computer, and is built into every Macintosh throughout the world.

This character set (see Figure 1-36) uses all character codes from $00–$FF, and includes uppercase versions of all of the lowercase accented Roman characters, a number of symbols, and other forms. A complete set of glyphs for all characters is available in most outline fonts, but not all characters are represented in the Apple bitmapped versions of Chicago, Geneva, New York, and Monaco.

The Standard Roman character set is an extended version of the original **Macintosh character set**, as described in Volume I of the original *Inside Macintosh.* It adds characters with codes from $D9–$FF, which are empty in the original Macintosh character set. Like the original Macintosh character set, the Standard Roman character set is an extended version of the **ASCII character set**. The ASCII character set, sometimes called *low ASCII*, is the traditional but limited character encoding for English-language computer systems. It uses character codes from $00–$7F only, and includes uppercase and lowercase letters, numerals, a few symbols, and a set of control (nonprinting) characters. The Standard Roman character set includes all the ASCII character codes and adds the characters (sometimes called *high ASCII*) with codes from $80–$FF.

The Standard Roman character set is implemented by the U.S. keyboard-layout resource (type = `'KCHR'`, ID = 0) and other Roman keyboard layouts. The Standard Roman character set and its sorting and formatting rules form a baseline which other script systems adopt, modify, or replace as their needs align with or diverge from the Roman conventions.

Figure 1-36 The Standard Roman character set

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	nul	dle	sp	0	@	P	`	p	A	ê	†	∞	¿	-	‡	
x1	soh	DC1	!	1	A	Q	a	q	Å	ë	°	±	¡	—	˙	Ò
x2	stx	DC2	"	2	B	R	b	r	Ç	ì	¢	≤	¬	“	‚	Ú
x3	etx	DC3	#	3	C	S	c	s	É	ì	£	≥	√	”	„	Û
x4	eot	DC4	$	4	D	T	d	t	Ñ	î	§	¥	ƒ	‘	‰	Ù
x5	enq	nak	%	5	E	U	e	u	Ö	ï	•	µ	≈	’	Â	ı
x6	ack	syn	&	6	F	V	f	v	Ü	ñ	¶	∂	∆	÷	Ê	ˆ
x7	bel	etb	'	7	G	W	g	w	á	ó	ß	∑	≪	◊	Á	˜
x8	bs	can	(	8	H	X	h	x	à	ò	®	∏	≫	ÿ	Ë	¯
x9	ht	em	)	9	I	Y	i	y	â	ô	©	π	…	Ÿ	È	˘
xA	lf	sub	*	:	J	Z	j	z	ä	ö	™	∫	nbsp	/	Í	·
xB	vt	esc	+	;	K	[	k	{	ã	õ	´	ª	À	¤	Î	˚
xC	ff	fs	,	<	L	\	l	|	å	ú	¨	º	Ã	<	Ï	¸
xD	cr	gs	-	=	M	]	m	}	ç	ù	≠	Ω	Õ	>	Ì	˝
xE	so	rs	.	>	N	^	n	~	é	û	Æ	æ	Œ	ﬁ	Ó	˛
xF	si	us	/	?	O	_	o	del	è	ü	Ø	ø	œ	ﬂ	Ô	ˇ

In Figure 1-36, note that each character code is represented by a two-digit hexadecimal number. The first digit is determined by the column, and the second by the row. For example, the character code for ¶ is $A6 (from column *Ax* at row *x6*).

Inconsistencies in Roman Character Encoding

For historical reasons, Roman character encoding has not always been consistent. The Roman script system in particular contains many fonts with unique glyphs that are not part of the Standard Roman character set. Since the character encoding is limited to 256 values, fonts such as Symbol, ITC Zapf Dingbats®, and other specialized fonts override the standard Roman character encoding.

For example, in the standard Roman character set $70 corresponds to lowercase "p", but it is the numeric symbol for pi ("π") in the Symbol font, an outlined square ("❒") in ITC Zapf Dingbats, and the musical symbol pianissimo for *play quietly* in the Sonata font. Hence, be aware that a Roman character code may have different interpretations in different fonts.

Furthermore, different variations of the Roman script system can have slightly different character encodings to allow for their slightly different character sets. This situation occurs only in the Roman script system; other script systems have uniform character encodings. The Roman character set and its variations are described in more detail in the appendix "Built-in Script Support" in this book.

Other 1-Byte Character Encodings

All 1-byte simple script systems have character encodings that can be thought of as simple substitutions for parts of the standard Roman character set. As noted previously, some encodings, such as Croatian or Turkish, replace or relocate relatively few characters, and are still considered Roman scripts.

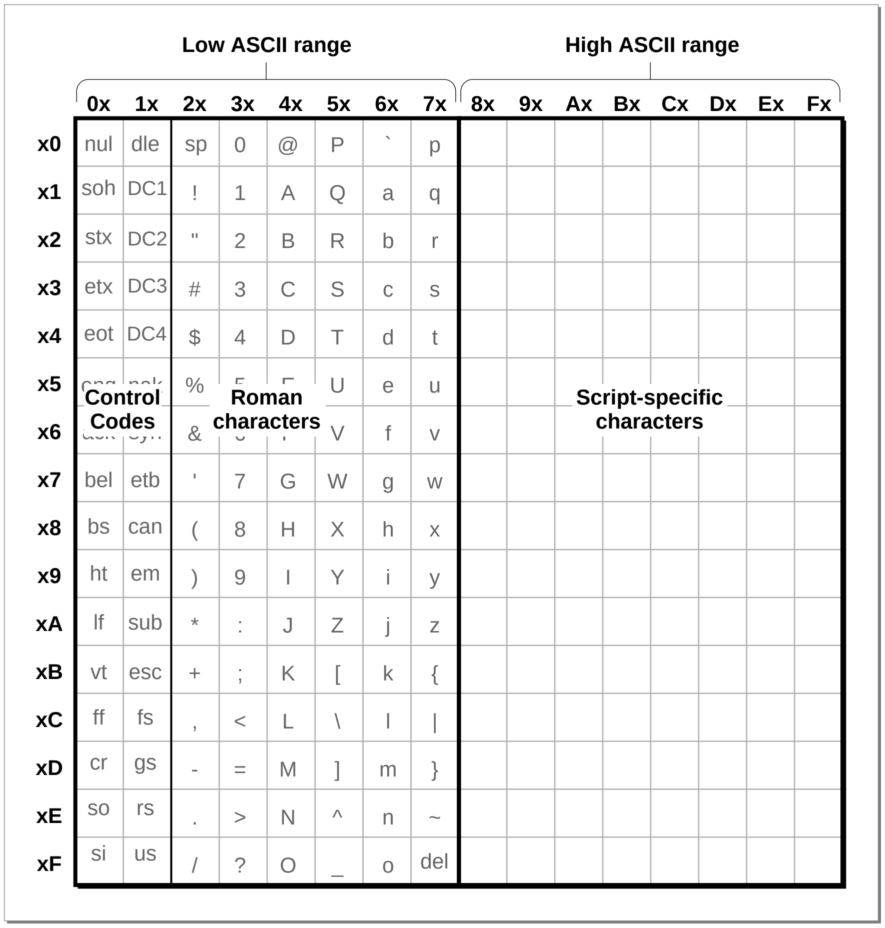
Other encodings for 1-byte simple script systems, such as Central European or Cyrillic, replace much of the high-ASCII range of the Standard Roman character set (code values from $80 to $FF) with a different alphabet.

The 1-byte complex script systems replace the same general range of Roman characters as do the 1-byte simple script systems, but they also define additional text forms in order to accommodate extensive use of ligatures or other contextual variations.

For all 1-byte script systems, the character sets include the standard low-ASCII control characters (code values from $00 to $1F) and Roman characters (code values from $20 to $7F). This allows users to enter Roman text, including western numbers, without having to switch script systems. It also allows applications to display low-ASCII Roman text regardless of the font in the current graphics port. It also means that control characters are interpreted as control characters in any script system. Figure 1-37 shows the general scheme of character encoding for 1-byte script systems.

Those 1-byte complex script systems that need more contextual forms than can fit in the high-ASCII range solve the problem through associated fonts and fonts with special glyph codes, rather than by changing any of the low-ASCII character encoding. See the discussion of associated fonts in "Font Handling" beginning on page 1-60.

Figure 1-37 Character encodings for 1-byte script systems


Low ASCII range
High ASCII range
Control Codes
Roman characters
Script-specific characters

	0x	1x	2x	3x	4x	5x	6x	7x	8x	9x	Ax	Bx	Cx	Dx	Ex	Fx
x0	nul	dle	sp	0	@	P	`	p								
x1	soh	DC1	!	1	A	Q	a	q								
x2	stx	DC2	"	2	B	R	b	r								
x3	etx	DC3	#	3	C	S	c	s								
x4	eot	DC4	$	4	D	T	d	t								
x5	enq	nak	%	5	E	U	e	u								
x6	ack	syn	&	6	F	V	f	v								
x7	bel	etb	'	7	G	W	g	w								
x8	bs	can	(	8	H	X	h	x								
x9	ht	em	)	9	I	Y	i	y								
xA	lf	sub	*	:	J	Z	j	z								
xB	vt	esc	+	;	K	[	k	{								
xC	ff	fs	,	<	L	\	l	|								
xD	cr	gs	-	=	M	]	m	}								
xE	so	rs	.	>	N	^	n	~								
xF	si	us	/	?	O	_	o	del								

2-Byte Character Encodings

Worldwide, the majority of script systems have character encodings that can fit within the limits set by the size of a byte, which permits up to 256 distinct characters. However, Asian scripts with ideographic characters, such as Chinese and Japanese, require thousands to tens of thousands of characters. The Korean script system, which is not ideographic, nevertheless requires at least 2,000 characters; furthermore, ideographic Chinese-derived characters are often included in Korean text.

To define that many characters requires 2-byte character codes. The Macintosh script management system is designed to handle 2-byte codes correctly. The use of script-aware routines permits your application to handle text without having to know whether each character code is 1 byte or 2 bytes, as long as the application allows for the possibility of 2-byte codes. Basically, that means not assuming that one byte equals one character, and not breaking or truncating text in the middle of a 2-byte character.

As with 1-byte script systems, the character encoding for each 2-byte script system includes the standard ASCII control characters (code values from $00 to $1F) and the low-ASCII Roman characters (code values from $20 to $7F) as a subscript. But in addition, a 2-byte script system may include a second set of Roman characters with 2-byte character codes, and character encodings for several other subscripts besides that of its native writing system. Figure 1-38 shows one example of a 2-byte encoding scheme.

IMPORTANT

2-byte scripts use a mixture of 1-byte and 2-byte encodings to represent characters. You cannot use the terms *byte* and *character* interchangeably, nor can you assume that every character is 2 bytes long. Obtaining character-type information about characters is discussed in the chapter "Script Manager" in this book. ▲

Japanese

Japanese is one of the most intricate writing systems in the world, containing four individual subscripts: Romaji (alphabetic Roman letters), Katakana and Hiragana (syllabic characters), and Kanji (ideographic characters). For example, the word *Japan* can be written in these four ways, as

Romaji,

N i h o n

Katakana,

ニホン

Hiragana,

にほん

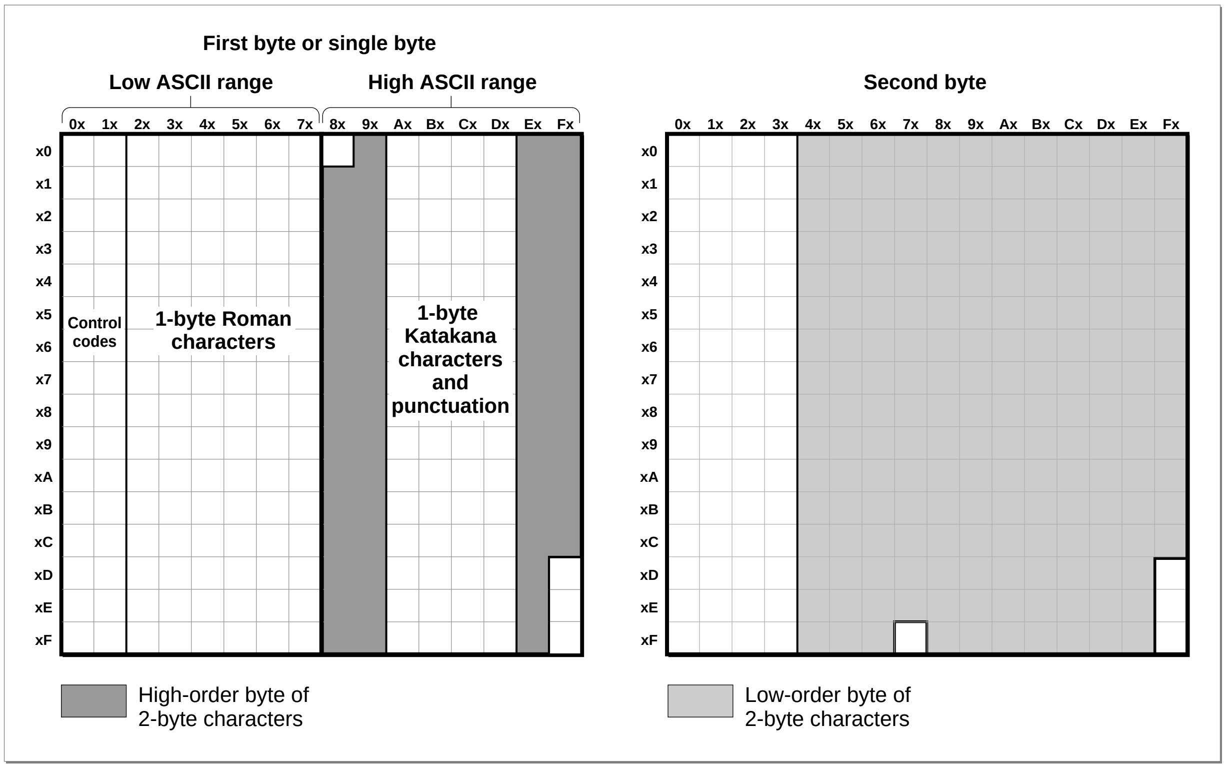
or Kanji:

日本

Romaji, Katakana, and Hiragana each have relatively few characters, but a minimal set of Kanji contains over 3,000 characters.

The Japanese character encoding can be thought of as an extension of a typical 1-byte character encoding. Control codes and low-ASCII Roman characters are in the range $00–$7F; script-specific 1-byte characters and the first bytes of 2-byte characters are in the range $80–$FF. Additional 256-byte tables contain the second bytes of the 2-byte characters.

Figure 1-38 Character encoding for a 2-byte script system (Japanese)


First byte or single byte
Low ASCII range
High ASCII range
Second byte
0x 1x 2x 3x 4x 5x 6x 7x 8x 9x Ax Bx Cx Dx Ex Fx
x0 x1 x2 x3 x4 x5 x6 x7 x8 x9 xA xB xC xD xE xF
Control codes
1-byte Roman characters
1-byte Katakana characters and punctuation
High-order byte of 2-byte characters
Low-order byte of 2-byte characters

In Figure 1-38, each 2-byte character code is represented by a four-digit hexadecimal number. The first two digits (the high-order byte) come from the *First byte* table, and specify which of the many *Second byte* tables contains the character. The second two digits (the low-order byte) come from the appropriate *Second byte* table. For example, the character code for 1 is $93FA (from column *Fx* at row *xA* in the *Second byte* table whose location is specifed by the value at column *9x* at row *x3* in the *First byte* table).

Chinese

The Macintosh script management system supports two separate Chinese script systems: Simplified Chinese and Traditional Chinese. Simplified Chinese consists of approximately 8,000 ideographic characters, about 2,000 of which have been simplified from their traditional presentation for ease of learning. Traditional Chinese consists of approximately 13,000 of the traditional Chinese ideographic characters, called Hanzi.

Simplified Chinese and Traditional Chinese use incompatible character encodings; the same character may have different character codes in the two scripts.

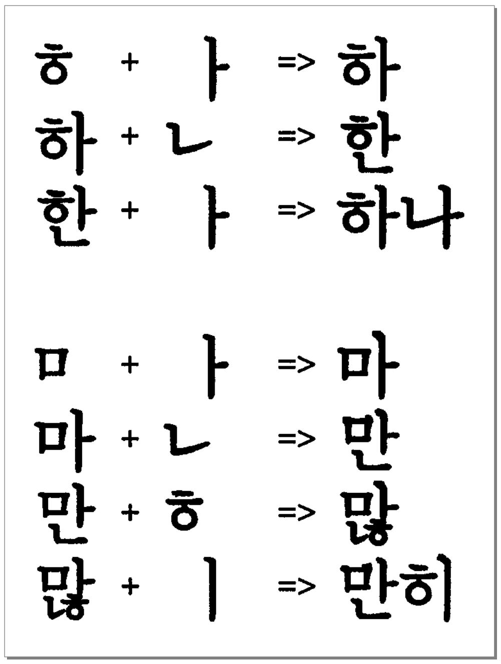
Korean

The Korean script system is based on characters of the Hangul subscript, devised in 1443. Chinese characters, called Hanja, are often mixed with Hangul, but their use is gradually declining. The Korean Standard Hangul Coding Scheme for Communications (KS5601) defines 2,350 Hangul characters for Korean writing, which form the basic character set of the Korean script system.

Hangul characters are syllabic blocks composed of component elements called **Jamo.** Jamo can be simple or double consonants and vowels. There are 24 simple Jamo elements and 27 double elements.

The first sound in a Hangul block is a simple or double consonant, the second is a simple or complex vowel, and the third (optional) sound is a simple, double, or complex consonant. Figure 1-39 shows an example. Each Hangul character (on the right) can have two or three elements (first sound and middle sound, plus optional last sound).

Figure 1-39 Constructing blocks (Hangul) from elements (Jamo) in Korean



Font Handling

As discussed under “Fonts” on page 1-44, a Macintosh font provides a specifically designed set of glyphs that implement the character set and other written forms that belong to a given script system. Fonts can be classified as 1-byte and 2-byte, and as bitmapped or outline. Some fonts provide plain (unstyled) glyphs, whereas others provide styled variations, such as bold or italic. This section summarizes some of the basic font issues to keep in mind when working with text, and especially multiscript text.

Font Availability and Selection

You cannot display text in a given script system without a font for that script system. A font is available only if its file resides in the Fonts folder within the System Folder, or if its resources are installed in the System file itself.

In terms of font availability and font selection for text, remember these points:

- The script management system uses font family ID (the ID number of the font resource of type `'FOND'`) to refer to fonts. That is the ID you supply as a parameter to text-handling calls. However, do not store font family ID numbers in your text files; store font names instead, and redetermine the ID numbers at run time with Font Manager calls. Font family IDs are not unique, and the system can renumber fonts between executions of your application.
- Every font family has an ID number in a range that identifies the script system it belongs to. Identifying the font family used to write text is equivalent to identifying the script system of that text. See Figure 1-35 on page 1-50 for an illustration of resource ID ranges. A text string in a font with a font family ID of 200 is interpreted as Roman text, while the same text string in a font whose ID is 17000 is interpreted as Chinese text and displayed accordingly. You can use the Script Manager to convert font IDs into script codes.
- Because the script management system uses the font associated with a given range of text to determine the script system of that text, store your text in such a way that, for each run of text, you track the font to be used to display it.
- Because the script system can be determined from just the font family ID, the Font Manager can use that information to substitute a font of the proper script system, even when an entire font family is missing.
- Because the Roman script system is present in all Macintosh systems, at least two Roman fonts are always available: 12-point Chicago and 12-point Geneva.

System Font and Application Font

Macintosh system software recognizes two special fonts that should always be present: the **system font** and the **application font.** The system font is the font used for menus, dialog boxes, and other messages to the user from the Finder or Operating System. The application font is the suggested default font for use by monostyled TextEdit and by applications that do not support user selection of fonts. In all unmodified Roman versions of Macintosh system software, the system font is 12-point Chicago and the application font is 12-point Geneva.

In all localized versions of Macintosh system software, whether Roman or not, the system font has a *special font designator* of 0, and the application font has a special designator of 1. These special designators are not actual font family resource ID numbers and cannot be used as such in Resource Manager calls; however, you can use them in place of a font family ID in the `txFont` field of the graphics port, and in text-related calls that take a font family ID, such as `FontToScript`. The system maps the special designators to the actual font family IDs for the system font and application font. You can use the Font Manager to determine the actual ID numbers of the system font and application font for any system script.

Remember these points about the system font and the application font, in relation to Chicago font, Geneva font, and the special designators:

- On localized versions of system software in which the system script is Roman, Chicago is the system font and it has a font family ID of 0. The special designator 0 also refers to Chicago font.
- When the system script is non-Roman, Chicago has a different font family ID (usually 16383), and the special font designator 0 refers to the system font for the non-Roman system script. On system software in which Japanese is the system script, for example, a value of 0 in the `txFont` field means the Osaka font, which has a font family ID of 16384.
- When the system script is Roman, Geneva is the application font and it has a font family ID of 3. The special designator 1 also refers to Geneva font.
- When the system script is non-Roman, Geneva has the same font family ID of 3, but the special font designator 1 refers to the application font for the non-Roman system script. On system software in which Thai is the system script, for example, a value of 1 in the `txFont` field means the Thonburi font, which has a font family ID of 26625.
- The actual font family ID of the system font is specified in the low-memory global variable `SysFontFam`; the actual font family ID of the application font is specified in the low-memory global variable `AppFontID`. You can get the actual font family ID of the system font or the application font by making Font Manager calls; see the chapter "Font Manager" in this book. You can also get the actual font family ID of the preferred system font or application font for a script system by making Script Manager calls; see the discussion of script variables in the chapter "Script Manager" in this book.

Perhaps the most common mistake developers make in adapting their applications to global markets is to assume that the application font is always Geneva. *Do not assume that different script systems have the same system and application fonts.*

Roman Characters and Associated Fonts

All Macintosh script systems include the low-ASCII Roman characters and control characters as part of their character sets. Most non-Roman fonts provide glyphs for those low-ASCII Roman characters. If the font itself does not contain those characters, the script system substitutes characters from an **associated font**—a Roman font that is associated with that script system—for character codes (mostly in the low-ASCII range) that the script system determines are Roman. Some contextual script systems must use associated fonts because they need more glyphs than can fit into the high-ASCII range normally available for native glyphs.

Note
A script system specifies the associated font for its system font and application font, but may allow the user to select a single Roman font to associate with all other fonts of the script system. ◆

In most cases your application does not have to account for associated fonts; glyphs from the associated font are substituted automatically when you draw text that contains Roman characters. However, keep in mind that font measurements (such as the results of the `GetFontInfo` and `FontMetrics` procedures) always account for the width and height characteristics of *both* the current font and the associated font. This can sometimes cause unexpected results, such as a line height that is greater than the current font's expected line height. The `GetFontInfo` procedure is described in the chapter "QuickDraw Text" in this book; font measurement and the `FontMetrics` procedure are further described in the chapter "Font Manager" in this book.

There are several other issues to keep in mind related to Roman characters and Roman fonts:

- Remember that the presence of Roman glyphs in displayed or printed text does not necessarily imply that they were created with a Roman font. The Text Utilities can help you locate Roman characters in a text buffer and explicitly change them to the Roman script system, if you wish.
- As noted on page 1-56, the Roman script system does not have a consistent character set across all fonts. For example, character codes in the Symbol font map to different glyphs from the same character codes in the Geneva font. Conversely, identical symbols can have different character codes in different fonts. The division sign (÷) is located at $D6 in the Helvetica font and $B8 in the Symbol font.
- Inconsistent character codes for symbols other than letters and numbers can also be a problem across script systems. For instance, in the Roman script system the division sign (÷) is located at $D6 in most fonts, whereas in the Arabic script system the division sign (÷) is at $9B.

Other Font Issues

In general, when drawing text, you set the font characteristics before you make a call, and the script management system makes sure that the font you specify is used. However, there are some issues and complications to keep in mind:

- If a particular size or styled variation (such as bold or italic) of a font is not available on the computer, the Font Manager can scale an existing size and QuickDraw can apply a style to an existing plain version of a font. Certain styles may be disabled in scripts where they are inappropriate. You can use the Script Manager to determine all of the valid styles for a given script system.
- The setting of the font force flag is controlled by the user when a script system that supports it is the system script. If the font force flag is `TRUE`, text written with a Roman font is considered instead to be text of the system script; any character codes corresponding to native characters of the system script are drawn in the system font rather than in the specified Roman font. If you do not want that to happen in your application, you must monitor the state of the font force flag and change it temporarily whenever necessary.
- The font force flag exists to permit multiple-language support by applications that expect a single font. It is only a partial solution to the problem. *Do not hardcode your application to require any single font.*

- If your application needs to have a font whose characters should never be interpreted as system script characters (for example, symbol fonts used for paint program palettes), you can assign the font an ID in the reserved range $7E00 to $7FFF (uninterpreted symbols) rather than in the Roman range. Then, even if the font force flag is set to `TRUE`, your symbols are not re-interpreted as system-script characters.
- When displaying characters as they are typed in by the user, you must make sure that the font for text display belongs to the same script system that is used for text input. See "Font Script and Keyboard Script" beginning on page 1-51.
- Many fonts—particularly those associated with non-Roman writing systems—do not draw legibly unless they are at least 12 point. However, you cannot assume that the system font size is always 12 point. Use QuickDraw, Font Manager, and Menu Manager calls to get the default size for the system font, default size for the current font, and required menu bar height for the system font.
- Do not assume that the application font exists in a 9-point size. Use the Script Manager to determine the application font family and size for legible small text.
- Diacritical marks (such as the acute accent over the "E" in "École") may extend above or below the normal limits for character height. The Font Manager allows you to either extend the spacing between lines or shrink the marked characters to make sure that the characters are not cut off at the top or bottom.
- If you use your own menu-definition (`'MDEF'`) resource to draw a Font menu in your application, be sure it can draw all font names correctly. It should use the font itself, or a font of the same script system, to display the font name. See the Menu Manager chapter of *Inside Macintosh: Macintosh Toolbox Essentials* for more information on creating menus.
- A 2-byte font can be very large; outline fonts for 2-byte script systems can contain single resources over 6 MB in size. Large numbers of 2-byte fonts can be a storage problem for the user. Furthermore, because the Resource Manager limits the size of a file's resource fork to 16 MB, it may be difficult to include 2-byte fonts with your application or document files.

Character Rendering and Text Display

The process of properly preparing characters for display is called *character rendering.* When QuickDraw draws a character, string, or line of text, it takes the stored character codes you supply it and processes them if necessary to take into account line direction, contextual substitution, or character reordering. It uses the rules of the font script (the script system of the current font of the active graphics port) to make these calculations. QuickDraw then gets the glyphs for the resulting characters from the Font Manager, and draws the glyphs in order on the screen, starting at the current pen location.

IMPORTANT

A fundamental assumption of the Macintosh script management system is that contextual analysis, character reordering, and the formation of ligatures should occur during the *display* of text, not its storage. That way the stored version of text can be much simpler; it contains only the basic characters of its writing system. Searching and other text-manipulation tasks are much more straightforward that way. It is the Macintosh script management system that has the job of handling differences between storage order and display order, and differences between stored codes and displayed glyphs. ▲

The 1-byte simple script systems and all 2-byte script systems currently have no individual character-rendering specifications; QuickDraw's built-in ability to draw characters sequentially in a given font, style, and size is sufficient.

The 1-byte complex script systems carry character-rendering information in line-layout tables in their encoding/rendering (`'itl5'`) resources. WorldScript I performs the rendering based on specifications in those tables.

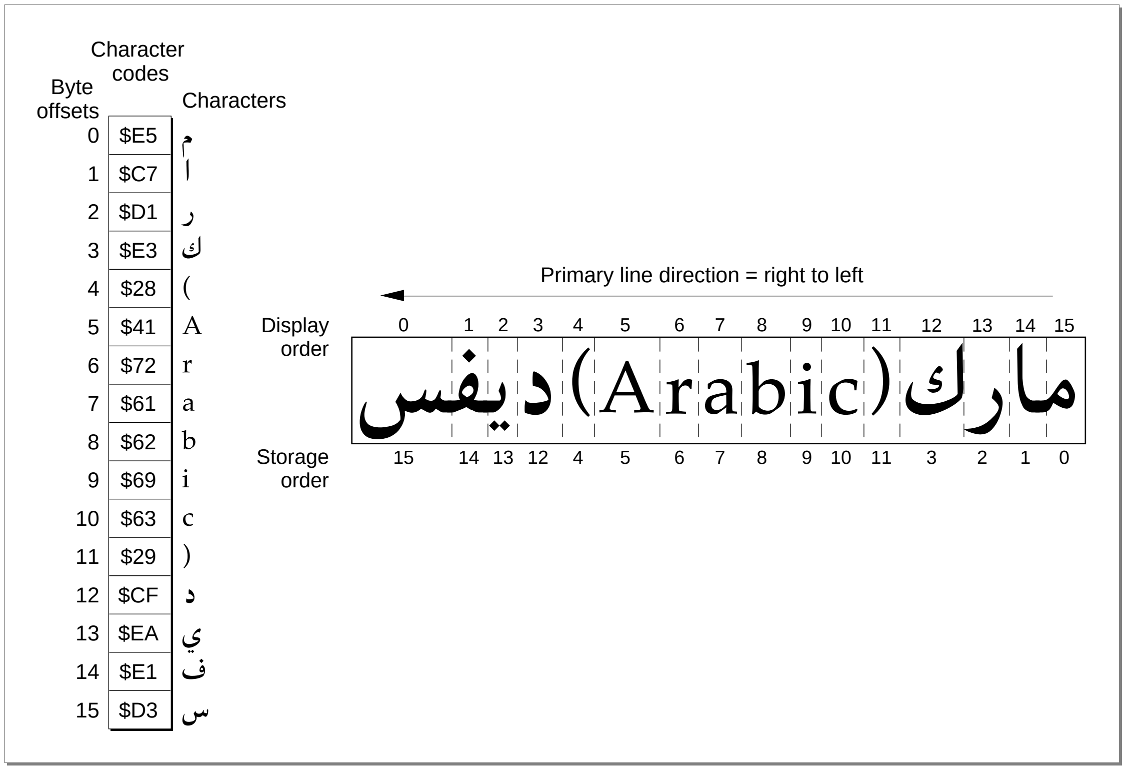
The section "Features of the World's Writing Systems" beginning on page 1-21 shows examples from writing systems that require the kinds of rendering abilities provided by the Macintosh script management system. Your application should not have to explicitly perform any of these tasks; you merely store character codes, and the script management system renders those characters properly whenever you need to display them.

Storage Order and Display Order

QuickDraw draws glyphs and lines of text from left to right only. This left-to-right orientation of QuickDraw is fundamental, and applies whether or not the text being drawn is meant to be read left-to-right or right-to-left. Each character is drawn with its origin (usually its left edge) placed at the current pen location, and after it is drawn QuickDraw moves the pen location rightward by the width of the glyph. Likewise, when QuickDraw draws a string of text, it keeps advancing the location as it draws, so that the pen location ends up at the right end of the string.

Display order is this left-to-right order in which QuickDraw draws glyphs on a display device. For example, QuickDraw draws a string of Hebrew text in reverse order from the way the string is read: the glyph for the last (= leftmost) character in the string is drawn *first,* and the glyph for the first (= rightmost) character in the string is drawn *last*. Figure 1-40 is an example showing a line of mixed Arabic and Roman text. The glyphs are drawn as shown, from left to right in the sequence labeled *Display order,* even though the primary line direction is right-to-left.

Figure 1-40 Storage order and display order



Storage order is the sequence of character codes in memory. The Macintosh script management system assumes that your application stores characters in the order in which they would be typed in—that is, with the first character code in a string at a lower address than subsequent character codes in that string. Storage order is different from display order for text with a right-to-left line direction.

In Figure 1-40, for example, the line of numbers labeled *Storage order* shows the byte offset in the buffer of the character for each glyph. Note that the glyphs for the Hebrew characters are drawn in reverse sequence from the order in which they are stored, whereas the glyphs for the Roman characters are drawn in the same sequence as their storage order.

If your application stores its text in the expected storage order, the script management system properly orders all characters within each style run that you draw.

Storage order can differ from display order not only in the sequence of individual characters within a run of text, but also in the order in which entire runs of text are drawn on the screen. See Figure 1-41 on page 1-67 for an example. If multiple scripts with different line directions occur on a single line, determining the order in which to draw the individual runs can be complex. The Macintosh script management system helps you with that determination; see the discussion of the `GetFormatOrder` procedure in the chapter “QuickDraw Text” in this book.

Line Direction and Alignment

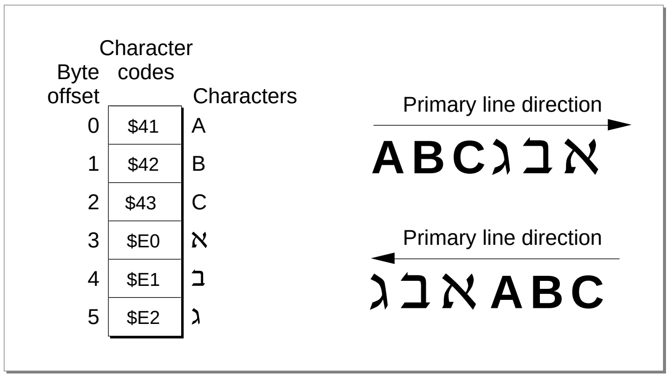
Writing systems exist with several different line directions, as shown in Figure 1-14 on page 1-24. The Macintosh script management system supports two of them: left-to-right (used for Roman and most other writing systems), and right-to-left (used for Arabic and Hebrew). As noted earlier in this chapter, Arabic and Hebrew systems are considered bidirectional rather than purely right-to-left because numbers and commonly intermixed foreign words are written from left to right. And although Japanese and Chinese are traditionally written vertically, the Japanese and Chinese script systems currently support only a left-to-right line direction.

The Macintosh script management system supports multiscript text, including text with mixed directions, in a single line. The layout, measurement, and drawing routines can help you correctly render text—even justifed text—from multiple script systems.

Primary Line Direction

When text with different line directions is mixed on a single line, the **primary line direction** is the principal, controlling direction for display of that text. The concept of primary line direction is important because it affects the order in which text elements are drawn. For example, suppose a block of Hebrew text follows (in storage order) a block of Roman text. If the primary line direction is left-to-right—equivalent to saying that the Hebrew text is embedded within a line of Roman text—the Hebrew text is drawn after and to the right of the Roman text. If the primary line direction is right-to-left—equivalent to saying that the Roman text is embedded within a line of Hebrew text—the Roman text is drawn after and to the right of the Hebrew text. Figure 1-41 illustrates the concept.

Figure 1-41 How primary line direction affects display order



Your application controls the primary line direction of its text by specifying it in parameters to certain text-layout calls such as the QuickDraw `GetFormatOrder` procedure. You can set your primary line direction independently of any system settings, but TextEdit and many text-processing applications tie their primary line direction to the current value of the system direction.

The **system direction** is a global setting, used by all parts of system software to control the alignment of text elements in dialog boxes, menus, and so on. TextEdit sets the primary line direction of its text to the system direction. Some script-aware routines assume that the primary line direction for the text they manipulate is equal to the system direction; see, for example, the description of the `CharToPixel` function in the chapter "QuickDraw Text" in this book.

System direction is determined by the value of the low-memory system global variable `SysDirection`. At startup, `SysDirection` is initialized to the line direction specified by the system's international configuration (`'itlc'`) resource. That value is commonly localized to correspond to the primary line direction of the system script, but if a bidirectional script system is enabled the user can control the system direction from the Text control panel; see "User Control of Script Settings" beginning on page 1-107.

Your application (and other applications) can also control the system direction with Script Manager routines. Do not simply assume a value for system direction.

The right-to-left primary line direction of bidirectional script systems has several further implications for program design. In working with bidirectional text, remember these points:

- Characters are read from right to left. Numerals are read from left to right. A word processor must therefore implement two sets of tabs and two ruler directions.
- Mathematical expressions are read from left to right in Hebrew and from right to left in Arabic. If in Hebrew one writes "6 + 4 = 10", in Arabic the same expression in the same order would be written "10 = 4 + 6".
- The concepts of **leading edge** and **trailing edge** of a glyph are important for mouse-down event testing, caret positioning, and highlighting. In left-to-right text, a glyph's leading edge is its left edge; in right-to-left text, a glyph's leading edge is its right edge. See "Caret Handling" beginning on page 1-74.
- Some punctuation marks and numerals from the Standard Roman character set are duplicated at different locations in bidirectional character sets in order to account for this. For example, the exclamation point (!) is at $21 in the Standard Roman character set, but Hebrew and Arabic add a second, right-to-left version of it, at $A1.
- Despite the fact that a single style run in a bidirectional script system can contain two directions of text, your application can treat it as a unit. See the note on bidirectional style runs on page 1-71.

Alignment

Alignment is the horizontal placement of lines of text with respect to the left and right edges of the text area or page. Text is typically left-aligned, right-aligned, centered, or justified—aligned to both the left and right margins. See Figure 1-15 on page 1-25.

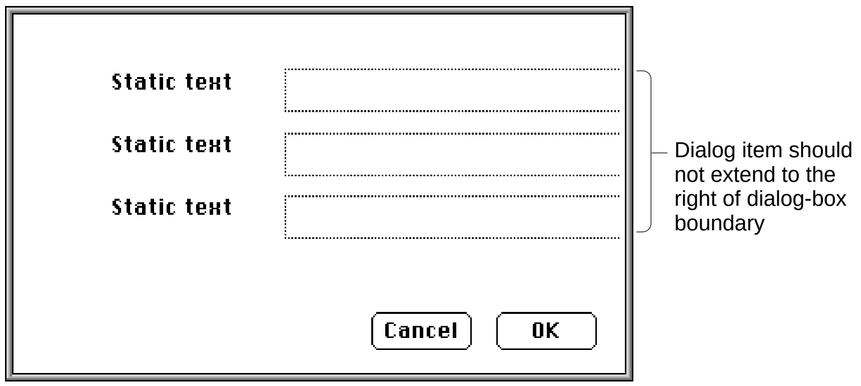
A script system's default text alignment usually follows its line direction. The system global variable `SysDirection`, which controls line direction, also controls the default alignment for text and other items in dialog boxes, alerts, and menus. For example, in Arabic system software (and in applications localized to the Arabic script system) menu items are right-aligned, and radio buttons and checkboxes are modified so that the boxes or buttons themselves are on the right. The user controls the system alignment by controlling the system direction with the Text control panel. See "User Control of Script Settings" beginning on page 1-107.

TextEdit uses the value of `SysDirection`, to set the default alignment for text in its windows.

You should anticipate that right alignment might occur in your application's text elements. Be sure to allow for it:

- Do not assume that, once you have measured the length of a line of text, you can always place it at the left margin. For right-aligned text, you need to indent the pen location from the left margin by an appropriate amount so that the right end of the text line falls on the right margin.
- Do not allow a text item in a dialog box to extend to the right of the dialog-box boundary; the right edge of a line of text in that item will be truncated if text is right-aligned. See Figure 1-42 for an illustration of this.

Figure 1-42 Dialog items truncated at dialog-box boundary


Static text
Static text
Static text
Cancel
OK
Dialog item should not extend to the right of dialog-box boundary

- When creating a column of checkboxes or radio buttons, make the text boxes all the same length. This ensures that when the line direction and alignment are reversed, the checkboxes or radio buttons remain correctly aligned.

If you are specifically formatting right-aligned text in a bidirectional script system, remember these additional alignment issues:

- Text is typically right-aligned. It breaks near the left margin and continues at the right margin of the following line. However, the "last" character on a line is not always the leftmost; see, for example, Figure 1-27 on page 1-31.
- Headers, footers, and footnotes are typically right-aligned.
- In a table or list, the first column is the rightmost.
- Line indentation is measured from the right margin.
- Odd pages are on the left in a book, and even pages are on the right. The inside front cover is on the right when a book is opened, and page 1 is on the left.

Justification

Justified text, which is aligned to both the left and right margins of the text area, is a special form of alignment that poses particular challenges to multiple-language formatting. The Macintosh script management system provides an entire set of routines for measuring, laying out, and drawing lines of justified text. See, for example, the descriptions of the `PortionLine` and `DrawJustified` routines, and the discussions of measuring and drawing lines of justified text in the chapter “QuickDraw Text” in this book.

Style Runs, Font Runs, Script Runs, Direction Runs

When QuickDraw draws a character or string of text, it examines the current graphics port record to determine how the text should be drawn. The font (and therefore the script system), the point size, and the style of the text are all determined by fields in the current graphics port.

This feature of QuickDraw has several consequences. First, it means that you must be sure to set the graphics port fields properly before calling QuickDraw. Second, it means that each call to QuickDraw must be restricted to a run of text that has uniform values for all those fields.

This finest division of the runs of text in your document is called a **style run.** A style run comprises the set of contiguous characters that all share the same font, size, and style characteristics. Because they share the same font, they naturally share the same script system. *The style run is the most important organizational unit for script-aware text handling,* and your application should always maintain style-run information for all its text. For many script-aware calls, you first set up the graphics port record appropriately and then make the call, passing it a single style run of text (or even less than a single style run, if the style run spans more than one line of text).

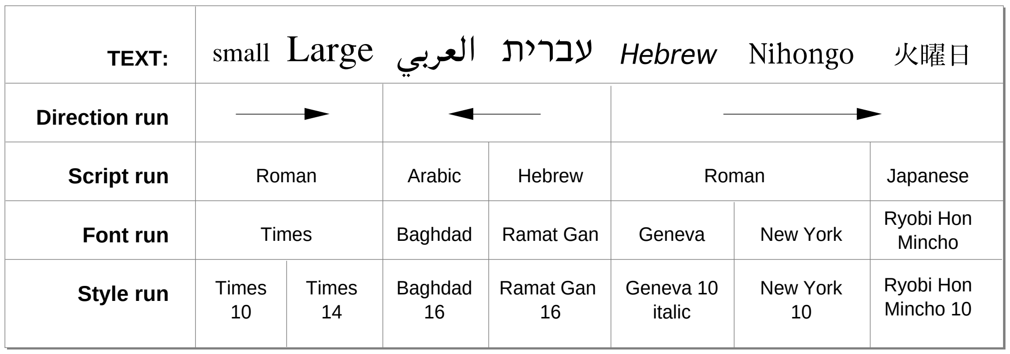
A larger division than style run is the **font run;** it consists of those characters that share the same font (and therefore the same script), but do not all share the same size or style attributes. You need not reset the `txFont` field of the graphics port between calls that involve text within a single font run.

The next larger division is the **script run;** it consists of all contiguous characters that belong to a given script system, regardless of their individual fonts. Within a script run, all the text’s formatting and text-manipulation specifications are constant; there is no need to load different resources or validate the existence of another script system between calls involving a single script run. If your application does not support multiple script systems, all of your text is a single script run.

The largest division is the **direction run.** A direction run consists of all contiguous characters with the same line direction, regardless of what script system they belong to. (However, see the note on bidirectional style runs page 1-71.) Within a direction run, the display order of characters and style runs has a very simple relation to their storage order. If all of your text consists of a single direction run, your text-layout tasks are simplified.

Figure 1-43 shows a line of text and its separation into style runs, font runs, script runs, and direction runs.

Figure 1-43 Style runs, font runs, script runs, and direction runs in text



TEXT:	small	Large	العربي	עברית	*Hebrew*	Nihongo	火曜日
Direction run	→		←		→		
Script run	Roman		Arabic	Hebrew	Roman		Japanese
Font run	Times		Baghdad	Ramat Gan	Geneva	New York	Ryobi Hon Mincho
Style run	Times 10	Times 14	Baghdad 16	Ramat Gan 16	Geneva 10 italic	New York 10	Ryobi Hon Mincho 10

Runs of Roman characters in text of a non-Roman script system are *not* necessarily considered separate style runs, and may be displayed (as Roman characters) in the non-Roman font of that script system. For greater formatting control, however, you may want to explicitly separate out those Roman characters into style runs of their own. You can use a Text Utilities routine to do so.

Bidirectional style runs

Bidirectional script systems have a unique concept of a style run. Numerals and Roman characters in a bidirectional style run have a different direction from the rest of the native text, but your application needn't consider them as separate style runs. The script management system handles all the special formatting, highlighting, and character location for you in these cases, so you can treat such a mixed-direction sequence just as you would any other single style run. ◆

Text Layout

Laying out lines of text—calculating how many characters fit on a line, determining the order of drawing of all the elements, performing all contextual formatting, and drawing the text—is a standard task in word processing. It can be a challenging task in a single language, but it is especially difficult to write a text-layout routine that is general enough to work with text in any script system. Even more complex is the laying out of text from several script systems in a single line. Add to that the complications of trying to draw justified lines of multiscript text, and the task can appear daunting.

The Macintosh script management system includes several groups of routines that ease the task by helping you write very generalized text-layout code that can handle multiscript lines of text, and can even justify those lines appropriately for the script systems involved. Routines from the Script Manager, the Text Utilities, and QuickDraw cooperate to analyze, arrange, format, measure, and draw the text.

There are two main principles that control how text layout occurs on the Macintosh:

- There is no system support for layout of more than a single line at a time. You are responsible for knowing where in memory your line starts and where on the screen to start drawing it.
- (Nearly) all text-layout routines operate on a single style run at a time. Therefore, to handle text with potentially multiple styles or scripts on a single line, you may need to call a routine repeatedly, once for each style run on the line.

Therefore, if a syle run extends beyond the boundaries of the current line, you call the routine for *only that portion of the style run that is on the line*. The part of a style run that exists on a single line is called a **text segment** in the chapter "QuickDraw Text" in this book.

In general, text layout involves taking the following steps, in order, for each line you intend to draw:

1. Starting with the buffer location of the first character on the line, and knowing the width of your display line in pixels, calculate the byte offset of the character at which to break the line. There are several ways to do this, using both QuickDraw and Text Utilities routines. The routines give proper results for any script system.
2. Determine the order in which to draw the individual style runs on the line, using a QuickDraw routine. If the line contains mixed-directional text, the left-to-right order in which you draw style runs may not be the same as the order in which they occur in memory. See, for example, Figure 1-41 on page 1-67.

   If you are drawing justified text, take these additional steps:

   - Eliminate trailing spaces at the end of the rightmost or leftmost (depending on the primary line direction) style run on your display line, so your justified text will line up properly. You can use a QuickDraw routine for this purpose (remember that a space character may not have the ASCII value $20 in a non-Roman script system).
   - Calculate the **slop value,** the extra amount of space that needs to be distributed throughout your line of text. Do that by measuring the total pixel width of all the style runs on the line and subtracting that from the display line width.
   - Calculate how to distribute that slop value among the style runs on your line, using a QuickDraw routine.
3. Position the QuickDraw pen both vertically and horizontally. The horizontal position must be at the left end of the text to be drawn on the line, regardless of the primary line direction. The vertical position is your responsibility; if you are drawing multiple lines in sequence, you can use QuickDraw or Font Manager routines to obtain font-height information to help you position the pen.
4. Draw the text, a style run at a time, using QuickDraw calls. For justified text, pass the amount of slop you calculated for each style run when you call the drawing routine for that run.

You can also use QuickDraw and Text Utilities calls to draw explicitly *scaled* multiscript text, in which the character are enlarged, shrunk, or distorted from their normal shapes; and you can even draw justified, scaled, multiscript text. For more information on text measurement and drawing, see the chapter "QuickDraw Text" in this book. For more information on line breaking, see the chapter "Text Utilities" in this book.

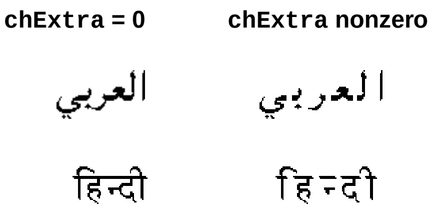
Remember these points when laying out and drawing lines of text:

- There are tables available that help you measure text before drawing it. The **global width table** is a table constructed in memory every time `FMSwapFont` is called; it can be used to calculate the pixel width of each glyph in a font, and it is helpful in determining line lengths. Each font family resource may have an optional width table with normalized glyph widths. Bitmapped fonts have tables that give the actual integer widths for their glyphs. For more information on these tables and on the `FMSwapFont` call, see the chapter "Font Manager" in this book.
- Don't break text into arbitrary chunks before formatting it for display: the second byte of a 2-byte character can be lost, or improper contextual formatting can result. If you need to truncate the displayed text at a location that is not a style run boundary or a valid line break or word boundary, use the QuickDraw clipping facility rather than truncating the string.
- Roman characters within style runs of a non-Roman font may display better if converted to the Roman script system and formatted as Roman text. You can use the Text Utilities to locate sequences of such characters.

If you are measuring the pixel width of a line of text before drawing it, keep these cautions in mind:

- Do not assume that a glyph for a given character code always has the same width. With certain scripts, using the Font Manager global width tables may give inaccurate results. The QuickDraw text-measuring routines return correct results for all script systems.
- Do not assume that specifying a fixed-width font in a graphics port always produces monospaced text. For example, the printed versions of some glyphs in some fixed-width fonts (such as "®" in Courier) have widths different from other glyphs in the font. Furthermore, when the Script Manager font force flag is set, the user might, for example, insert a wide Japanese character within a line of Monaco text. See the description of the font force flag in the chapter "Script Manager" in this book.
- Some characters, such as diacritical marks, may have zero width. A zero-width character should never be divided from the previous character in the text when you partition text. When truncating a string to fit into a horizontal space, the correct algorithm is to truncate from the end of the string toward the beginning, one character at a time, until the total width is small enough. This prevents cutting text before a zero-width character. You can also call Text Utilities functions to perform the truncation correctly.

- Do not set the `chExtra` field of the graphics port to a nonzero value with text containing connected glyphs or text that may include zero-width characters. Diacritical marks are placed incorrectly in relation to their base characters, and connected glyphs have white space inserted improperly, like this:



Caret Handling

By standard word-processing convention, the **selection range** is the sequence of zero or more characters—contiguous in memory—where the next editing operation is to occur. A selection range of zero characters is called an **insertion point.**

Highlighting a selection range and marking the insertion point both involve converting offsets of characters in a text buffer into pixel positions on a display device. In multiscript text, expecially text that has mixed line directions and contextual formatting, this can be a complex task.

The Macintosh script management system provides a routine that helps you draw carets properly for text in any combination of script systems. The QuickDraw function `CharToPixel` returns the onscreen pixel position corresponding to a given offset in your text buffer. The function returns the horizontal offset (in pixels) from the left margin of the text you pass it to the proper caret position corresponding to the character at the specified byte offset in your text buffer.

Caret and cursor

By convention in this book, the **caret** is defined as the blinking bar that marks the insertion point in text. The **cursor,** on the other hand, is the arrow, I-beam, spinning disk, or other small icon that marks screen position and moves with the mouse. ◆

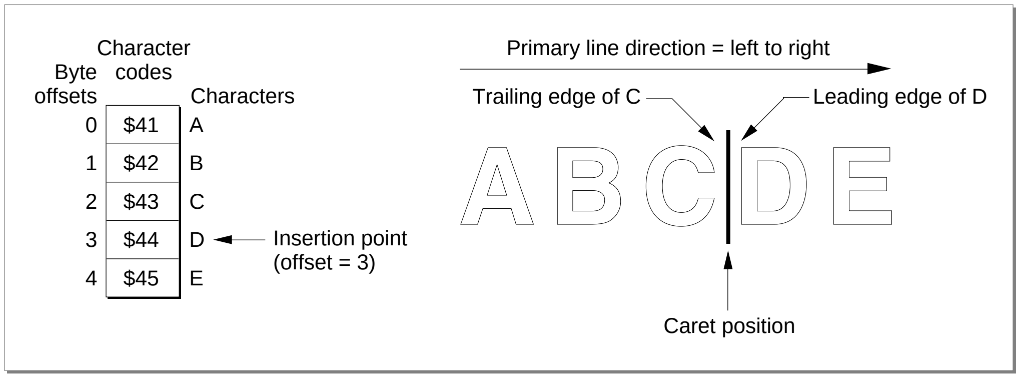
This section discusses the conventions underlying the relationship of text offset to caret position. For more information on conversion of text offset to screen position, see the description of the `CharToPixel` function in the chapter “QuickDraw Text” in this book.

The Caret

A **caret position** is a location on the screen that corresponds to an insertion point in memory. It lets the user know where in the text file the next insertion (or deletion) will occur. A caret position is always *between* glyphs on the screen, usually on the leading edge of one glyph and the trailing edge of another. The **leading edge** of a glyph is the edge that is encountered first when reading text of that glyph's script system; the **trailing edge** is opposite from the leading edge. In left-to-right text, a glyph's leading edge is its left edge; in right-to-left text, a glyph's leading edge is its right edge.

In most situations for most text applications, the caret position is on the leading edge of the glyph corresponding to the character at the insertion point in memory; see Figure 1-44. When a new character is inserted, it displaces the character at the insertion point, shifting it and all subsequent characters in the buffer forward by one character position. (That shift may be one or two bytes, depending on the size of the inserted character.)

Figure 1-44 Caret position and insertion point

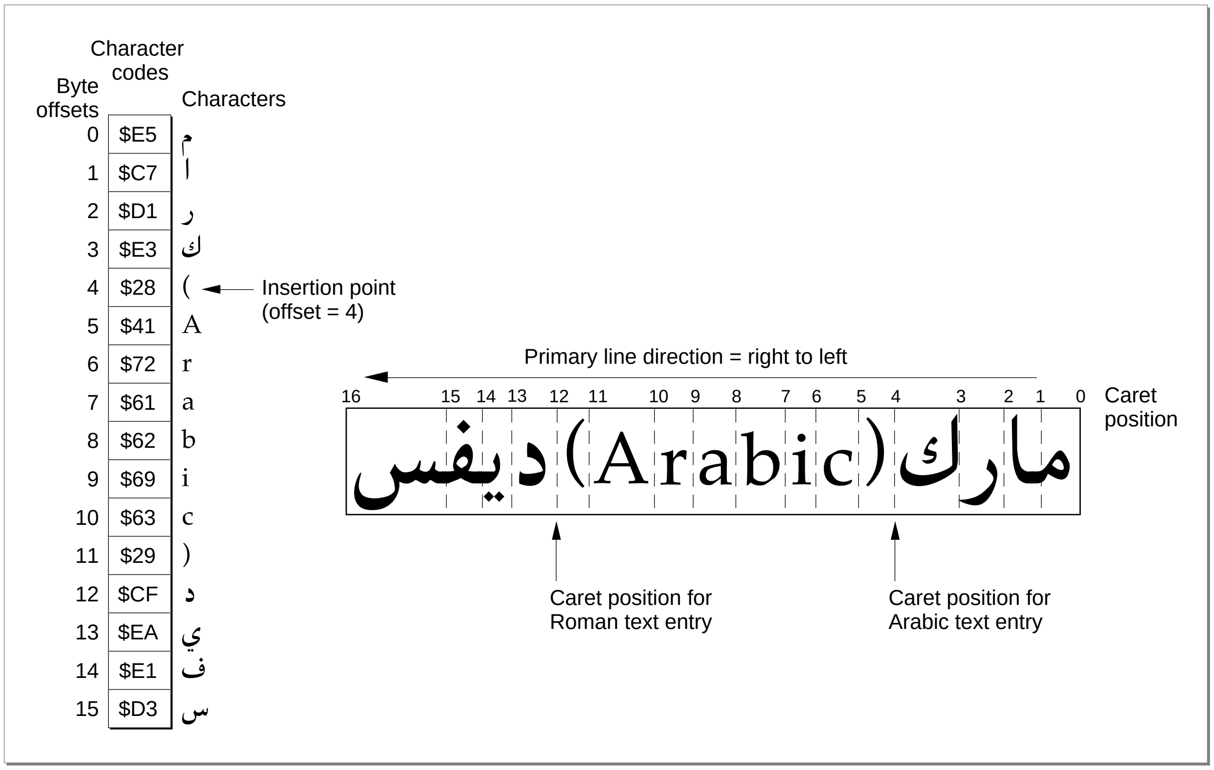


The caret position is unambiguous in text with a single line direction. In such a case, the caret position is on the trailing and leading edges of characters that are contiguous in the text buffer; it thus corresponds directly to a single offset in the buffer. This is not always the case in mixed-directional text, as described next.

Caret Positions at Direction Boundaries

In determining caret position, an ambiguous case occurs at direction boundaries because the byte offset in memory can map to two different glyph positions on the screen—one for text in each line direction. In Figure 1-45, for example, the insertion point is at byte offset 4 in the buffer. If the next character to be inserted is Arabic, the caret should be drawn at caret position 4 on the screen; if the next character is English, the caret should be drawn at caret position 12.

Figure 1-45 Caret positions at direction bondaries



The Macintosh script management system codifies this relationship between text offset and caret position as follows:

- For any given offset in memory, there are two potential caret positions:
  - □ the *leading edge* of the glyph corresponding to the character *at that offset*
  - □ the *trailing edge* of the glyph corresponding to the *previous (in memory) character*

  (The first and last characters of a text segment are special cases; see the discussion of the `CharToPixel` function in the chapter "QuickDraw Text" in this book.)
- In unidirectional text, the two caret positions coincide: the leading edge of the glyph for one character is at the same location as the trailing edge of the glyph for the previous character. In Figure 1-44, the offset of 3 yields caret positions on the leading edge of "D" and the trailing edge of "C", which are the same unambiguous location.
- At a boundary between text of opposite directions, the two caret positions do not coincide. Thus, in Figure 1-45, for an offset of 4 there are two caret positions: 12, on the leading edge of "("; and 4, on the trailing edge of ",,". Likewise, an offset of 12 yields two caret positions (also 12 and 4, but on the edges of two different glyphs).

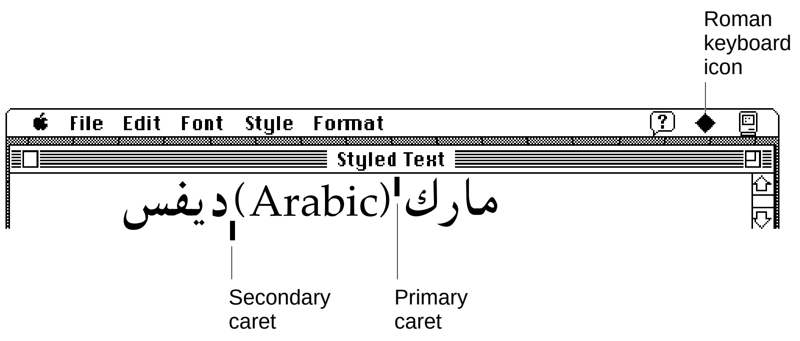
  At an ambiguous character offset, the current line direction (the presumed direction of the *next character to be inserted*) determines which caret position is the correct one:
  - □ If the current direction equals the direction of the character at that offset, the caret position is the leading edge of that character's glyph. In Figure 1-45, if Roman text is to be inserted at offset 4 (occupied by a Roman character), the caret position is on the leading edge of that character's glyph ("(")—that is, at caret position 12.
  - □ If the current direction equals the direction of the previous (in memory) character, the screen position is on the trailing edge of the glyph corresponding to that previous (in memory) character. In Figure 1-45, if Arabic text is to be inserted at offset 4, the caret position is on the trailing edge of the glyph of the character at offset 3 (",,")—that is, at caret position 4.

Two common approaches for drawing the caret at direction boundaries involve the use of a dual caret and a single caret. A **dual caret** consists of two lines, a high caret and a low caret, each measuring half the text height; see Figure 1-46. The high caret is displayed at the primary caret position for the insertion point; the low caret is displayed at the secondary caret position for that insertion point. Which position is primary, and which is secondary, depends on the primary line direction:

- The **primary caret position** is the screen location associated with the glyph that has the same direction as the primary line direction. If the current line direction corresponds to the primary line direction, inserted text will appear at the primary caret position. A **primary caret** is a caret drawn at the primary caret position.

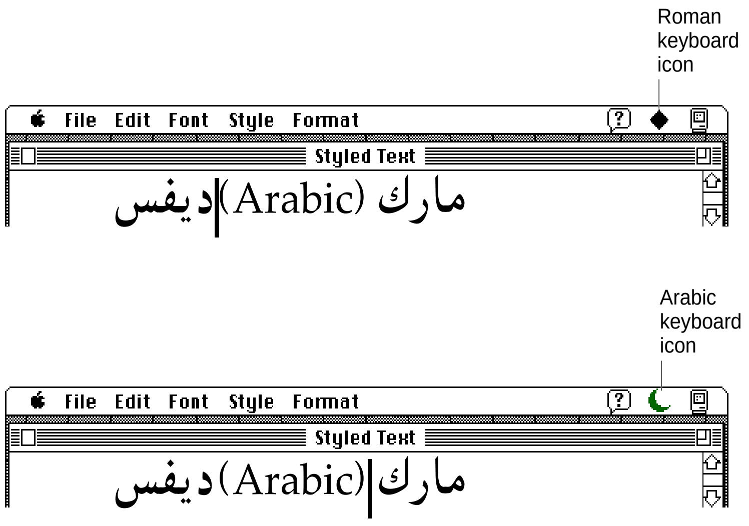
- The **secondary caret position** is the screen location associated with the glyph that has a different direction from the primary line direction. If the current line direction is opposite to the primary line direction, inserted text will appear at the secondary caret position. In Figure 1-46, the display of the Roman keyboard icon shows that the current line direction is not the same as the primary line direction, so the next character inserted will appear at the secondary caret position. A **secondary caret** is a caret drawn at the secondary caret position.

Figure 1-46 Dual caret at direction boundaries in mixed-directional text



A **single caret** (or **moving caret**) is simpler than a dual caret; see Figure 1-47. It is a single, full-length caret that appears at the screen location where the next glyph will appear. At direction boundaries, its position depends on the keyboard script. At a direction boundary, the caret appears at the primary caret position if the current line direction corresponds to the primary line direction; it appears at the secondary caret position if the current line direction is opposite to the primary line direction. The moving caret is also called a *jumping caret* because its position "jumps" between the primary and secondary caret positions as the user switches the keyboard script between the two text directions represented.

Figure 1-47 Single carets at direction boundaries in mixed-directional text



The script management system permits the user to select a preference between dual carets and a single (moving) caret; your text application should support both. TextEdit employs both types of carets; see the chapter "TextEdit" in this book.

Caret Movement With Arrow Keys

Most text applications allow the user to move the caret through displayed text with the arrow keys. In general, using the Right or Left Arrow key should move the caret uniformly right or left, regardless of the line direction of the text in which the caret appears. To do this means that your application needs to take the current line direction into consideration, rather than simply advancing the insertion point through the text buffer in response to presses of, say, the Right Arrow.

When the caret moves through a direction boundary (or any style run boundary) in response to a series of arrow keypresses, you need to set the keyboard script (and graphics port settings) to match the characteristics of the text that the caret is in. By convention, you should change the keyboard script and port characteristics after the caret has *passed* the boundary, not when it first reaches it.

For a discussion of how TextEdit handles the complications that occur at direction boundaries and within runs of bidirectional text, see the chapter "TextEdit" in this book.

Highlighting

When displaying a selection range, an application typically marks it by **highlighting,** drawing the glyphs in inverse video or with a colored or outlined background. As part of its text-display tasks, your application is responsible for knowing what the selection range is and highlighting it properly—as well as for making the necessary changes in memory that result from any cutting, pasting, or editing operations involving the selection range.

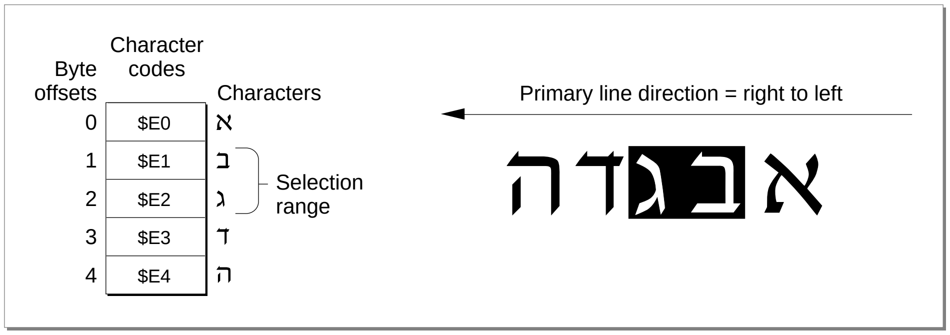
Discontinuous selection

A selection range as defined in this book always consists of characters contiguous in memory. Some word processors allow for **discontinuous selection,** in which the characters that constitute the selection range are not necessarily contiguous in memory. You can think of discontinuous selection as the simultaneous existence of several selection ranges of the type described here. Discontinuous selection is not discussed further in this book. In particular, keep in mind that the discontinuous *highlighting* shown in this section is not an example of discontinuous selection; all selection ranges shown here are single, contiguous ranges in memory. ◆

Unidirectional Text

In text with a single line direction, the selection range always appears on screen as a continuous range of highlighted glyphs; see Figure 1-48.

Figure 1-48 Highlighting a selection range in unidirectional text



The Macintosh script management system measures the limits of highlighting rectangles in terms of caret position. Thus, in Figure 1-48, in which the selection range consists of the characters at offsets 1 and 2 in memory, the ends of the highlighting rectangle correspond to caret positions for offsets 1 and 3. It’s equivalent to saying that the highlighting extends from the leading edge of the glyph for the character at offset 1 to the leading edge of the glyph for the character at offset 3.

Highlighting for word selection

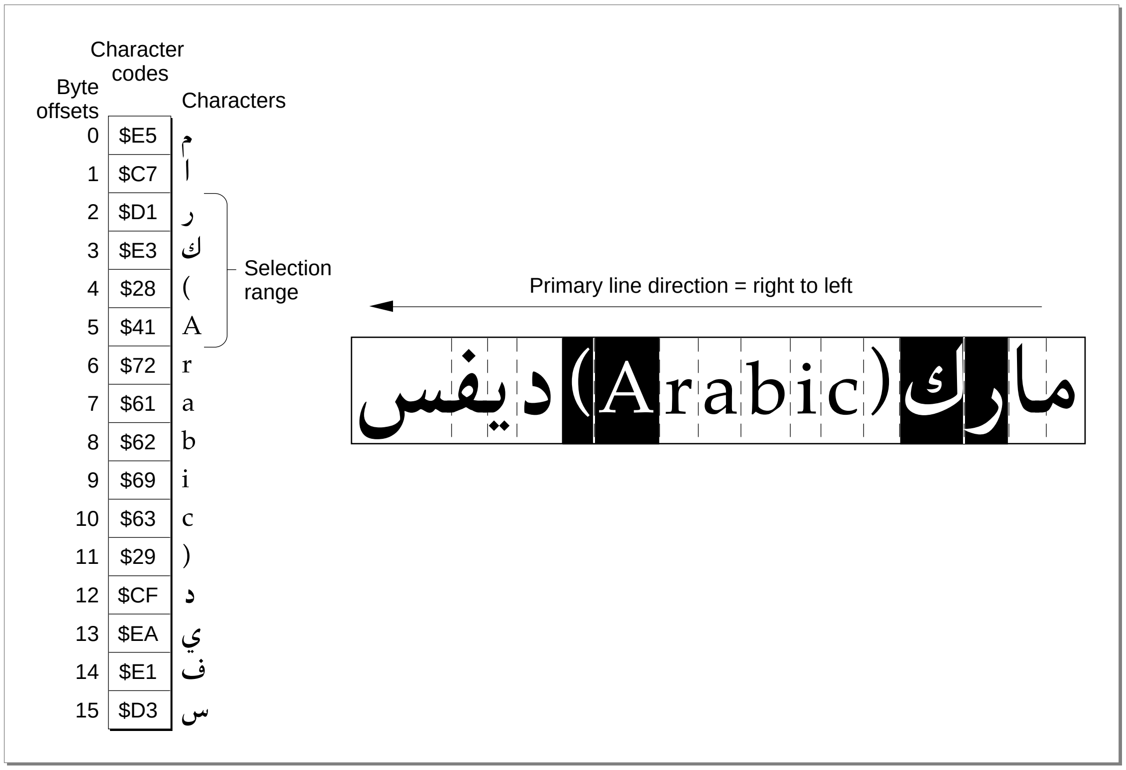
If your application supports word selection by double-clicking, it involves three steps. First, use a QuickDraw call to locate the offset in memory corresponding to the double-click. Second, use a Text Utilities call to locate the offsets of the word boundaries on either side of the double-click. Third, use QuickDraw calls to determine the boundaries of the rectangle to highlight. ◆

Mixed-Directional Text

If the displayed text has mixed direction runs, the selection range may appear as discontinuous highlighted text. This is because the characters that make up the selection range are always contiguous in memory, but characters that are contiguous in memory may not be contiguous on screen.

Figure 1-49 is an example of text whose selection range consists of a contiguous sequence of characters in memory, whereas the highlighted glyphs are displayed discontinuously.

Figure 1-49 Highlighting a selection range in mixed-directional text



Byte offsets	Character codes	Characters
0	$E5	م
1	$C7	ا
2	$D1	ر
3	$E3	ك
4	$28	(
5	$41	A
6	$72	r
7	$61	a
8	$62	b
9	$69	i
10	$63	c
11	$29	)
12	$CF	د
13	$EA	ي
14	$E1	ف
15	$D3	س

In describing the boundaries of the highlighting rectangles in terms of caret position, note that for Figure 1-49 it is not possible to simply say that the highlighting extends from the caret position of offset 2 to the caret position of offset 6. Using the definitions of caret position given earlier, however, it is possible to define it as two separate rectangles, one extending from offset 4 to offset 2, and another extending from offset 12 to offset 6 (assuming for the ambiguous offsets—4 and 12—that the current text direction equals the primary line direction).

The QuickDraw function `HiliteText` makes those kinds of calculations and is especially useful for determining the correct caret positions when highlighting a selection range in mixed-directional text. See the discussion of `HiliteText` in the chapter "QuickDraw Text" in this book for more details.

Converting Screen Position to Text Offset

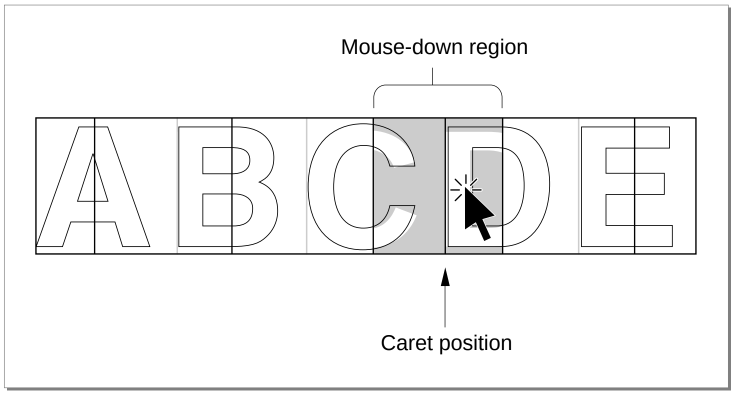
Caret handling and highlighting, as just discussed, require conversion from text offset to screen position. But that is only half the picture; it is just as necessary to be able to convert from screen position to text offset. For example, if the user clicks the cursor within your displayed text, you need to be able to determine the offset in your text buffer equivalent to that mouse-down event. You can then use that information to set the insertion point or selection range.

The script management system does most of this work for you. It provides routines that convert a screen position to the byte offset of a character code in memory (and vice versa); those routines function correctly with multiscript text, even text that has been rendered with ligatures and contextual forms.

Determining the character associated with a screen position requires first defining the caret position associated with a given screen position. Once that is done, the previously defined relationship between caret position and text offset can be used to find the character.

Figure 1-50 shows the cursor positioned within a line of text at the moment of a mouse click. A mouse-down event can occur anywhere within the area of a glyph, but the caret position that is to be derived from that event must be an infinitesimally thin line that falls between two glyphs.

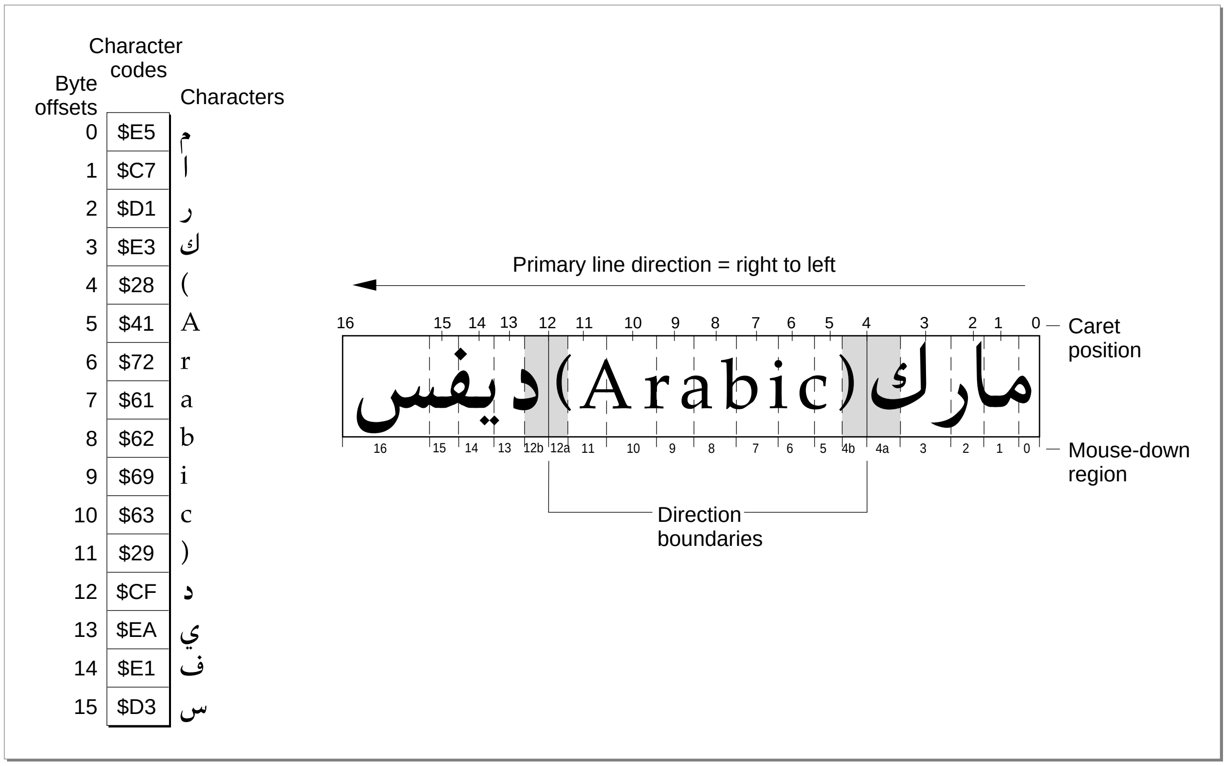
Figure 1-50 Interpreting caret position from a mouse-down event



A line of displayed glyphs is divided by the script management system into a series of mouse-down regions. A **mouse-down region** is the screen area within which any mouse click will yield the same caret position. For example, a mouse click that occurs anywhere between the leading edge of a glyph and the center of that glyph results in a caret position at the leading edge of that glyph. For unidirectional text, mouse-down regions extend from the center of one glyph to the center of the next glyph (except at the ends of a line), as Figure 1-50 shows. A mouse click anywhere within the region results in a caret position between the two glyphs.

At line ends, and at the boundaries between text of different line directions, mouse-down regions are smaller and interpreting them is more complex. As Figure 1-51 shows, the mouse-down regions at direction boundaries extend only from the leading or trailing edges of the bounding glyphs to their centers. Note that the shaded part of Figure 1-50 is a single mouse-down region, whereas *each* of the shaded parts of Figure 1-51 is two mouse-down regions.

Figure 1-51 Mouse-down regions and caret positions in mixed-directional text



Byte offsets	Character codes	Characters
0	$E5	م
1	$C7	ا
2	$D1	ر
3	$E3	ك
4	$28	(
5	$41	A
6	$72	r
7	$61	a
8	$62	b
9	$69	i
10	$63	c
11	$29	)
12	$CF	د
13	$EA	ي
14	$E1	ف
15	$D3	س

How do mouse-down regions relate to text offset? Referring to Figure 1-51, and remembering that the primary line direction is right-to-left, consider the two mouse-down regions 4a and 12a:

- A mouse click within region 4a is associated with the trailing edge of the Arabic character “,,”. In response, your application might make the keyboard script Arabic, draw a primary caret (or single caret) at caret position 4, and place the insertion point at offset 4 in the buffer, to insert Arabic text following “,,”. (If you are drawing a dual caret, the secondary caret should be at caret position 12, which also corresponds to an insertion point at offset 4 in the buffer.)
- A mouse click within region 12a is associated with the leading edge of the Roman character “(”. In response, your application might make the keyboard script Roman, draw a secondary caret (or single caret) at caret position 12, and place the insertion point at offset 4 in the buffer, to insert Roman text preceding “(”. (If you are drawing a dual caret, the primary caret should be at caret position 4, which also corresponds to an insertion point at offset 4 in the buffer.)

Thus mouse clicks in two widely separated areas of the screen can lead to an identical caret display and to a single insertion point in the text buffer. One, however permits insertion of Roman text, and the other Arabic text, and the insertions occur at different screen locations.

Mouse clicks in regions 4b and 12b in Figure 1-51 would lead to just the opposite situation: a primary caret at caret position 12, a secondary caret at caret position 4, and an insertion point at offset 12 in the text buffer. Either Roman text would be inserted after the Roman character “)”, or Arabic text would be inserted before the Arabic character “œ”.

The QuickDraw function `PixelToChar` helps you make these calculations; it returns the byte offset in your text buffer corresponding to the character associated with a particular distance (in pixels) from the left margin of the displayed text. It even handles the special cases of pixel locations outside (to the left or right of) the margins of the displayed text. For more information on conversion of screen position to text offset, see the description of the `PixelToChar` function in the chapter “QuickDraw Text” in this book.

Printing

At the application level, printing on the Macintosh computer is not fundamentally different from drawing to the screen. A printer is considered a display device, and your application prints by creating a printing graphics port (a graphics port with a few extra fields for printing), setting the port’s fields, and drawing in the port with calls to QuickDraw. General procedures for printing are described in the Printing Manager chapter of *Inside Macintosh: Imaging.* However, printing text, and especially contextual text, can pose extra challenges.

A very common complication results from the difference in resolution and pixel size between screen and printer. QuickDraw measurements are theoretically in terms of **points,** which are nominally equivalent to screen pixels at normal resolution. High-resolution printers have very much smaller pixel sizes, although printer drivers are expected to take this into account so that the same QuickDraw calls will produce text lines of the same width on the screen and on a printer. Nevertheless, this higher resolution, and the fact that printers can use different fonts from those used for screen display, can result in some loss of fidelity from the screen to the printed page:

- QuickDraw places text glyphs on the screen at whole screen-pixel intervals, whereas a high-resolution printer has much smaller pixels and can therefore provide much finer placement on the printed page. If your application specifies the use of fractional glyph widths, the spacing of the text on the screen can be awkward but it more accurately reflects the optimum layout of the printed text. Alternatively, specifying integer glyph widths gives more pleasing screen results because the characters are drawn with regular pixel spacing, but the results on the page can be typographically unacceptable. See the discussions of fractional glyph width in the chapters “QuickDraw Text” and “Font Manager” in this book for more information.
- Printer drivers attempt to reproduce faithfully the text formatting as drawn by QuickDraw on the screen, including keeping the same intended character spacing, line breaks, and page breaks. However, because printers can have resident fonts that are different from the fonts that QuickDraw uses, because the drivers may handle text layout somewhat differently than QuickDraw, and because font metrics do not always scale linearly, fidelity may not always be achieved. Typically, identical line breaks and page breaks can be maintained, but character spacing can be noticeably different.

Other complications result from the fact that high-resolution printers use *deferred printing,* in which the document to be printed is first converted into a spool file in picture-file format, and it is the picture file rather than the original document that is printed. This can result in loss of certain display features that picture files do not support, such as the following:

- The `grayishTextOr` transfer mode cannot be used for printing. See the discussion of transfer modes in the chapter "QuickDraw Text" in this book.
- You cannot pass `DrawText` (or `StdText`) more than 255 bytes of text at a time when printing. `DrawText` and `StdText` are documented in the chapter "QuickDraw Text" in this book.

Some of the most difficult problems result from the fact that printer drivers replace the QuickDraw bottleneck routines `StdText` and `StdTxMeas` (by changing the `grafProcs` field of the printing graphics port) to allow printing to function with QuickDraw calls, whereas certain script systems use different modifications (trap patches) to those same routines to perform contextual formatting. Printer drivers that print from spool files can then interact with QuickDraw in several ways that may cause complications:

- Some drivers call QuickDraw twice: once to create a spool file for printing, and once again to unwind the spooling. If the text is contextually transformed during spooling, the transformation must not be repeated during unwinding.
- Some drivers may not call QuickDraw at all, meaning that necessary contextual transformations might not be made at all.
- Some drivers may call QuickDraw re-entrantly, such as when displaying a status message during printing.

To avoid these problems, printer drivers should call the Script Manager Print Action routine whenever they change the `grafProcs` field. For more information on the Print Action routine, see the discussion on writing device drivers in *Inside Macintosh: Devices*.

To accommodate the special contextual formatting needs of 1-byte complex script systems, WorldScript I patches the QuickDraw routines `StdText`, `StdTxMeas`, `MeasureText`, and the Font Manager procedure `FontMetrics`. There are Script Manager routines that allow you to modify or replace those patches if your text has additional needs not met by the WorldScript I routines. To allow for the extra complications that may occur during printing, WorldScript I allows you to define separate entry points or even separate routines for printing as opposed to screen display. See the discussion on replacing a script system's default routines in the chapter "Script Manager" in this book, and the description of WorldScript I in the appendix "Built-in Script Support."

Text Input

Typically, your application accepts text input from the user through the keyboard. The Macintosh script management system allows you to accept text input in any script system, and to switch easily among input script systems.

Keyboard input is a complex process that involves conversion of hardware keypresses to software **raw key codes,** then to **virtual key codes,** and finally to character codes. Subsequent display of those input characters on the screen involves conversion of character codes to the glyphs of a font, and the drawing of those glyphs on the screen. As noted under "Separation of Tasks" beginning on page 1-4, text input and text display are completely independent of each other.

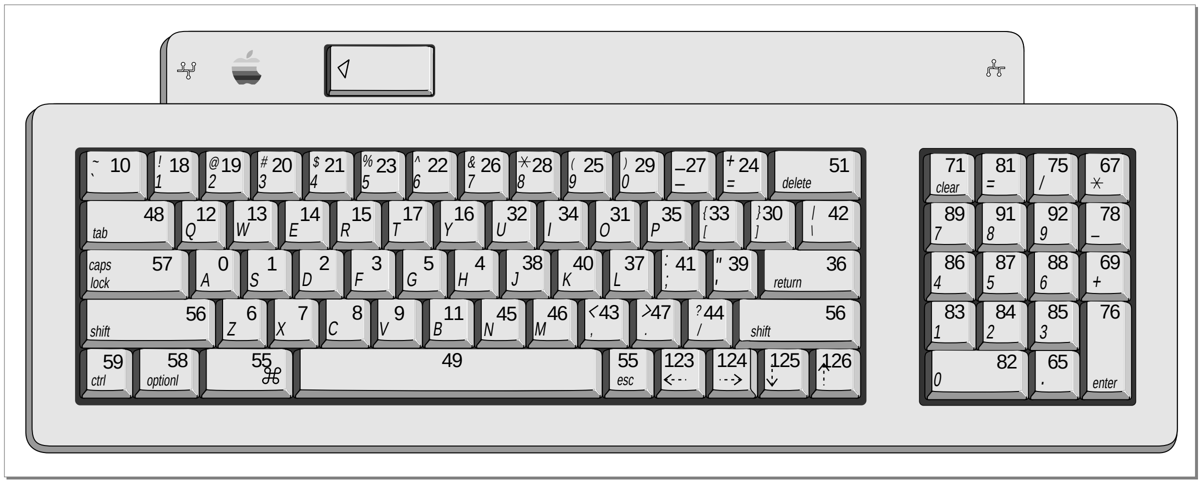
The conversion of keypresses to character codes is complex because the Macintosh computer has to support many different physical keyboards and many script systems. The conversion of raw key codes to virtual key codes accommodates the spectrum of keyboards; the conversion of virtual key codes to character codes accommodates the spectrum of script systems.

For 1-byte script systems, characters are generated directly from keypresses. For 2-byte script systems, the large number of characters makes direct keyboard input impractical; those systems provide input methods to make text input more convenient.

Keyboards and Key Translation

Every Macintosh keyboard has a specific physical arrangement of keys. An example is shown in Figure 1-52. The figure shows the physical arrangement of keys on the domestic (U.S.) layout of the Apple Keyboard II. It also shows the virtual key codes produced when each key is pressed, as well as the character generated (for U.S. system software) by each key.

Figure 1-52 Apple Keyboard II (domestic layout)



Other keyboards produce a similar set of virtual key codes, although the keys and their codes may be arranged differently. Apple supports at least 13 separate physical keyboards, listed in the appendix "Keyboard Resources" in this book. All can produce a set of hardware-independent virtual key codes, which translate directly into the characters of any script system. That process is called **key translation.**

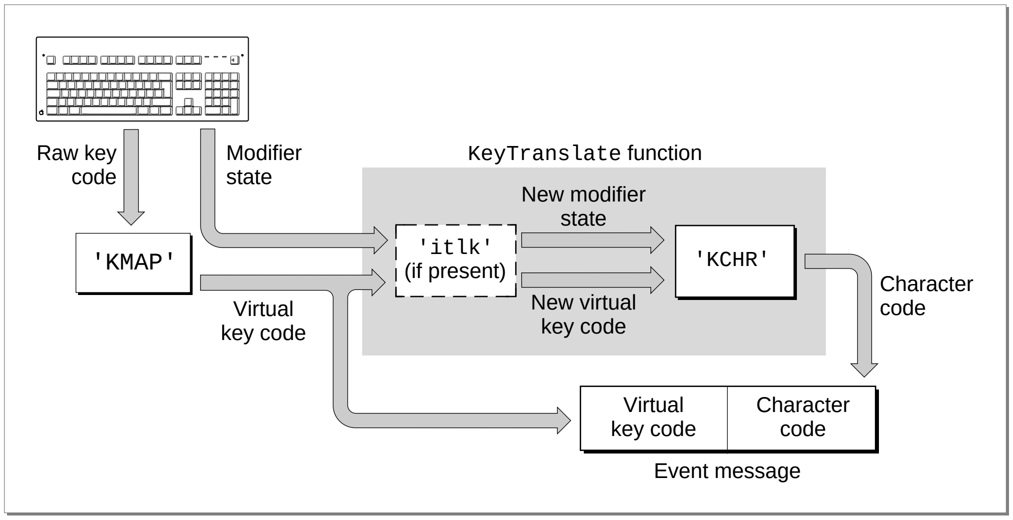
As far as the application is concerned, text input for all keyboards and for all script systems is hardware-independent. Except for a few minor hardware-specific characteristics, the function of the keyboard is completely determined by a script system's keyboard-layout (`'KCHR'`) resources. Tables within the keyboard-layout resource specify the characters produced by each key in combination with each modifier key (Command, Shift, Caps Lock, Control, and Option).

Figure 1-53 illustrates the process of key translation. A keypress initially produces a raw key code. The keyboard driver uses the hardware-dependent key-map (`'KMAP'`) resource to map the raw key code into a hardware-independent virtual key code and to set bits indicating the state (up or down) of the modifier keys. It then calls the Event Manager `KeyTranslate` function.

If the optional key-remap (`'itlk'`) resource is present, `KeyTranslate` uses it to remap certain key combinations on certain keyboards before performing additional processing. The key-remap resource transforms this information based on which keyboard is in use. It reintroduces hardware dependence because certain writing systems, languages, and regions need subtle differences in layout for specific keyboards. Generally, the key-remap resource affects only a few keys.

The `KeyTranslate` function then uses the current script's keyboard-layout resource to map the virtual key code and modifier state into a character code. `KeyTranslate` returns the character code, and the keyboard driver posts the key-down event into the event queue. The application receives the original virtual key code and a character code in the `message` field of the event record, and modifier-key information in the `modifiers` field of the event record.

Figure 1-53 Key translation



The `KeyTranslate` function is described in the chapter “Event Manager” in *Inside Macintosh: Macintosh Toolbox Essentials.* For additional information on the `KeyTranslate` function and the keyboard-layout resource, see the appendix “Keyboard Resources” in this book.

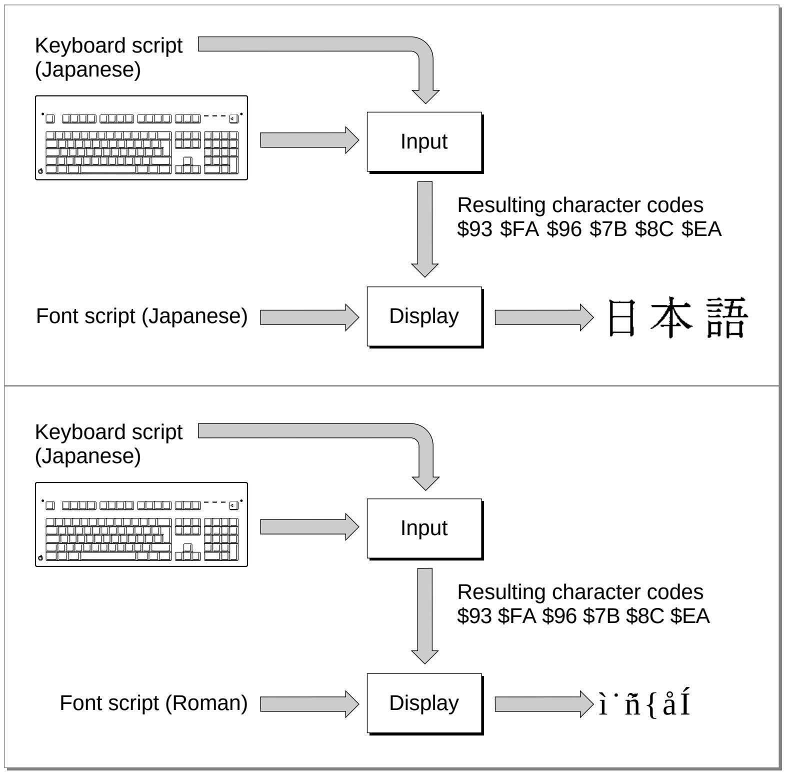
Dead keys

The keyboard-layout resource also handles dead keys, by means of additional subtables. A **dead key** is a key combination that has no immediate effect, but sets a state that affects the results of the next keypress (typically, the generation of one or two characters). Dead keys are commonly used to generate accents and accented characters. Dead-key processing is discussed in more detail in the appendix “Keyboard Resources” in this book. ◆

Font and Keyboard Synchronization

Whenever your application displays text as it is being entered at the keyboard, it needs to keep the font script coordinated with the keyboard script (see "Font Script and Keyboard Script" beginning on page 1-51). The upper half of Figure 1-54 shows an example of font and keyboard synchronization with the user entering the characters for *Nihongo* when the font script corresponds to the keyboard script, which is Japanese. The lower half of Figure 1-54 provides an example of the characters that are displayed when the user enters the same characters when the font script does not match the keyboard script. If the two scripts don't match, the results are meaningless to the user.

Figure 1-54 Font script and keyboard script synchronization



You use the Script Manager `KeyScript` procedure to set the keyboard script when, for example, the user chooses a new font from your Fonts menu or when the user clicks in an area of text that has a font different from the current one. The Operating System automatically changes the keyboard script (or keyboard layout or input method) when the user chooses a new one from the Keyboard menu (see Figure 1-62 on page 1-106). When that happens you need to set the font script to equal the keyboard script.

The Operating System also automatically changes the keyboard script (or keyboard layout or input method) when the user presses certain key combinations, as specified by the keyboard-swap (`'KSWP'`) resource. When that happens you should set the font script to equal the keyboard script.

You can force a particular keyboard layout to be used with your application by using the Script Manager to define the default keyboard layout for a script system and then calling `KeyScript`.

For more information on setting the font script and keyboard script, see the discussion on making keyboard settings and the description of the `KeyScript` procedure in the chapter "Script Manager" in this book. For more information on the keyboard-swap resource, see the appendix "Keyboard Resources" in this book.

Handling Keyboard Equivalents

Many applications support keyboard commands or keyboard equivalents to menu commands. This can be a problem in a multiscript environment. Be careful of these issues in the keyboard equivalents that you allow:

- Avoid keyboard equivalents that use the Space bar in combination with the Command key and other modifier keys. Command–Space bar and Command–Option–Space bar are already commonly used for switching among script systems and keyboard layouts. See the discussion of the `KeyScript` procedure in the chapter "Script Manager" and the description of the keyboard-swap resource in the appendix "Keyboard Resources" in this book.
- When the Command key is pressed, some characters—such as the period or question mark—cannot be produced on certain keyboard layouts. To make Command-key handling work in these cases, it may be necessary to use the virtual key code to determine which character code *would have been produced* if the Command key had not been pressed. For more information, see the discussion of special uses for the `KeyTranslate` function in the appendix "Keyboard Resources" in this book.
- If your application extends the set of standard Macintosh modifier-plus-key combinations for specific purposes, your keyboard equivalents might not function properly in all script systems. Be sure to supply alternative methods—such as menu or dialog-box items—for gaining access to such features.

Input Methods

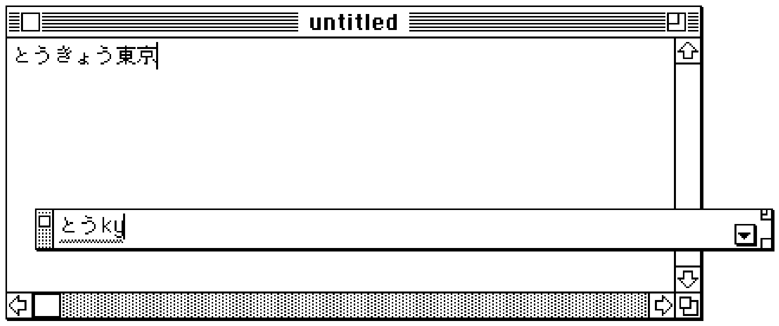
Script systems for ideographic writing systems such as Japanese cannot simply use a larger keyboard or multiple dead keys for effective text input. The sheer numbers of their characters demand a more complex solution, such as providing ways to convert phonetic text into ideographic text and vice versa. Most script systems with large character sets provide for the complex parsing of phonetic sequences to produce ideograms and character clusters.

Automatic conversion of phonetic glyphs into final representations is performed by an input method. For example, the Japanese script system supplements the keyboard by providing software for transcribing Kana (phonetic Japanese) into ideographic Kanji. Each Kanji character can correspond to more than one possible Kana sequence, and vice versa. The input method must grammatically parse sentences or phrases of Kana text (which has no word separations), and select the best combination of Kanji and Kana characters to represent that text.

Entry and Conversion

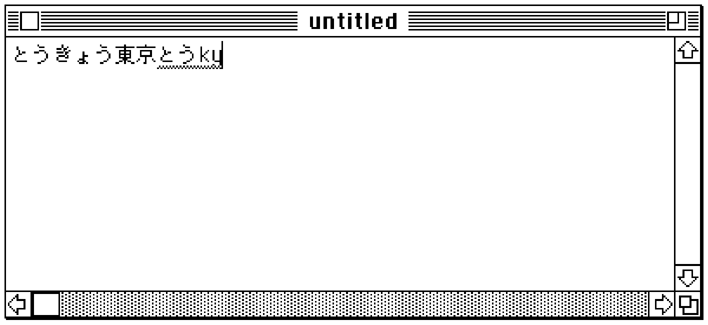
When a user types a character, one kind of input method opens a window (called a **floating input window** or **bottomline input window**) at the bottom of the screen for text entry; see Figure 1-55. In Japanese, the user can type using either Roman or Kana characters. When the converted glyphs are in the window, the user can freely cut and paste or convert them to any of the other subscripts.

Figure 1-55 Bottomline input window for Japanese input method



The Text Services Manager supplies an interface for input methods that use **inline input.** In inline input, the user types directly into an **active input area** within a document, as shown in Figure 1-56. Conversion then occurs within the active input area.

Figure 1-56 Active input area (underlined) for inline input



Input methods are often extended so that glyphs may be converted in extremely precise ways. For example, in the Japanese script system, when the text is converted to Kanji, the user has the option of changing any individual phrase: lengthening it, shortening it, or selecting different possible interpretations. All of the commands that perform these changes have both mouse and keyboard equivalents. Once the user presses the Return key, the text is entered as if it had been typed directly from the keyboard.

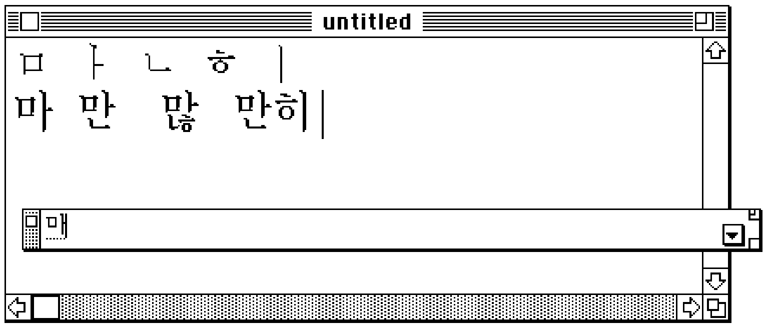
Differences Among Script Systems

In Japanese and Chinese input methods, the principal conversion is from Roman or other phonetic input to **Han** (Chinese) characters. In Japanese the input can be Romaji (Roman), Hiragana (phonetic), or Katakana (phonetic); the output is Kanji (Chinese characters). In Chinese the input can be **Pinyin** (Roman) or **Zhuyinfuhao** (phonetic; also called **Bopomofo**), and the output is Hanzi (Chinese characters). Chinese and Japanese use a semi-automatic conversion to Han characters that requires user confirmation.

The Korean script system's input method converts from Jamo (phonetic) to Hangul (clusters of Jamo). Transcription to Hanja (Chinese characters) is optional. Furthermore, the Korean input method uses a completely automatic conversion from Jamo to Hangul; user confirmation is not required.

Figure 1-57 illustrates the process of constructing Hangul from Jamo during bottomline input. Note that an added Jamo can appear in various positions (beside, beneath, and so on) relative to the Jamo or Hangul that it is added to.

Figure 1-57 Bottomline input in Korean



To gain the greatest acceptance worldwide, your application should support text input, and preferably inline input, in 2-byte script systems. For additional information on input methods, inline input, and input-method dictionaries, see the chapters "Text Services Manager" and "Dictionary Manager" in this book.

Text Manipulation

The Macintosh Operating System and Macintosh script management system implement certain script features transparently. For example, your application may not need to know that its dialog boxes can accept Japanese text. However, if your application actually manipulates the text of any language—as any word processor certainly does—it needs access to text-handling information that varies from script system to script system.

For example, to perform word selection and line-breaking, your application may need routines to determine word boundaries in any language. To sort text, it may need routines that sort acording to language-specific rules, and possibly also routines that perform case conversion or strip diacritical marks according to language-specific rules. Do not assume that all languages or regions have the same rules or conventions; use Text Utilities and Script Manager routines to handle different conventions.

Note that the user can affect which of the available script-system resources are used to control text manipulation such as sorting, number formatting, and date and time formatting. See “User Control of Script Settings” beginning on page 1-107.

This section discusses routines that perform a variety of script-aware text manipulations including sorting strings; formatting dates, times, and numbers; analyzing characters; searching and modifying text; and finding word boundaries and line breaks. Most of these topics are described more fully in the chapters “Text Utilities” and “Script Manager” in this book.

Note
The script management system does not address all possible localizable text issues. There is other information, not covered in a script’s international resources, that may vary from locale to locale—such as formats for addresses, postal zone codes, and telephone numbers. You should place all such information in resources for ease of localization. ◆

Sorting Strings

Comparing strings can be an intricate operation that involves subtle issues. Even for English, determining the sorting order cannot be done by a simple table look-up or comparison of character-code values. Furthermore, sorting rules vary not just among script systems but among the individual languages or regions within a script system.

Every script system—and every language-specific variation of a script system—has information specifying how its text is to be sorted. That information is in the script's string-manipulation (`'itl2'`) resource. The Text Utilities provides routines for comparing two strings for sorting purposes. Some routines work with Pascal strings, others with generalized **text strings** (defined by pointer and length). Some are script-aware, some are not. The script-aware routines take into account the sorting rules of the current script system or any script system that you explicitly specify, and can address these sorting factors:

- primary and secondary sorting order
- expansion and contraction of characters
- ignorable characters
- case-conversion and stripping of diacritical marks

Other special cases, such as expansion of abbreviations that requires dictionary lookup, may be beyond the capability of the script management system.

In sorting lists of strings that may be from more than one script system, keep these points in mind:

- If you are sorting strings from different script systems into a single list, the ordering relationship among the scripts as well as the sorting rules within each script are important. The script-sorting (`'itlm'`) resource that is part of system software contains tables that define the sorting relationship among all defined script systems. Text Utilities functions use that information to help you create a sorted list of strings in more than one script system.
- If you are sorting strings from different languages within a single script system, you may or may not want to sort the strings into groups by language. If you do, you can determine the ordering relationship between the languages from the the script-sorting resource. Text Utilities functions use that information to help you create a sorted list of strings in more than one language.
- If you need to sort strings in exactly the same way that the Macintosh file system does, there are Text Utilities routines that perform that type of sorting. The sorting order is fixed, and it is independent of any script system or language. It should be used only for operations internal to your application, not for user display of sorted filenames or other text strings.
- Uppercase characters and diacritical marks affect sorting and searching, and conventions for their handling vary among script systems and languages. Text Utilities routines allow you to sort according to the rules of each script or language, and to take into account or disregard case and diacritical marks.

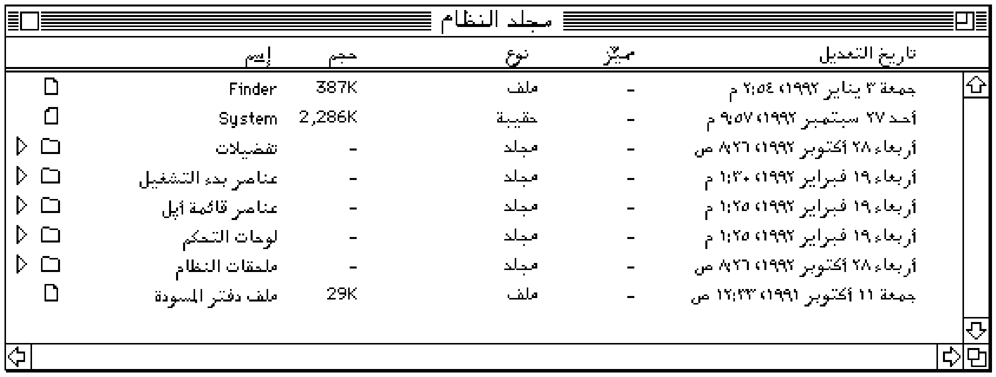
Formatting Dates, Times, Numbers, and Symbols

Dates, times, numbers, and symbols are common types of specialized strings whose formats vary widely around the world. Each script system defines how its times, dates, numbers, and other symbols are to be defined and formatted in its numeric-format (`'itl0'`), long-date-format (`'itl1'`), and tokens (`'itl4'`) resources.

Dates and Times

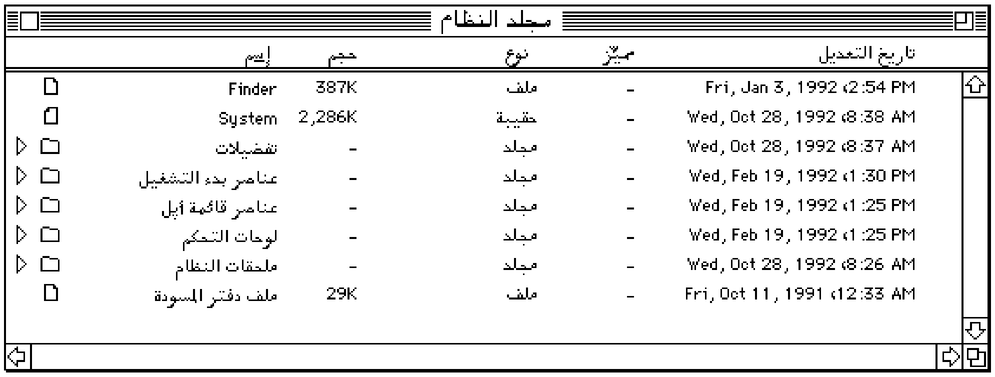
Figure 1-58 shows two different Finder displays of the same filenames and modification dates. The upper display uses Arabic date formats, Arabic month names (with theGregorian calendar), Arabic numerals, and a right-to-left primary line direction. The lower display is exactly the same, except with U.S. date formats, English month names, western numerals, and a left-to-right primary line direction. (The changes were made with control panel selections; see “User Control of Script Settings” beginning on page 1-107.)

Figure 1-58 Filenames and dates in Arabic and U.S. formats (Arabic system script)



مجلد النظام

إسم	حجم	نوع	ميّز	تاريخ التعديل
Finder	387K	ملف	-	جمعة ٣ يناير ١٩٩٢، ٢:٥٤ م
System	2,286K	حقيبة	-	أحد ٢٧ سبتمبر ١٩٩٢، ٩:٥٧ م
تفضيلات	-	مجلد	-	أربعاء ٢٨ أكتوبر ١٩٩٢، ٨:٢٦ ص
عناصر بدء التشغيل	-	مجلد	-	أربعاء ١٩ فبراير ١٩٩٢، ١:٣٠ م
عناصر قائمة أبل	-	مجلد	-	أربعاء ١٩ فبراير ١٩٩٢، ١:٢٥ م
لوحات التحكم	-	مجلد	-	أربعاء ١٩ فبراير ١٩٩٢، ١:٢٥ م
ملحقات النظام	-	مجلد	-	أربعاء ٢٨ أكتوبر ١٩٩٢، ٨:٢٦ ص
ملف دفتر المسودة	29K	ملف	-	جمعة ١١ أكتوبر ١٩٩١، ١٢:٣٣ ص



مجلد النظام

إسم	حجم	نوع	ميّز	تاريخ التعديل
Finder	387K	ملف	-	Fri, Jan 3, 1992 ،2:54 PM
System	2,286K	حقيبة	-	Wed, Oct 28, 1992 ،8:38 AM
تفضيلات	-	مجلد	-	Wed, Oct 28, 1992 ،8:37 AM
عناصر بدء التشغيل	-	مجلد	-	Wed, Feb 19, 1992 ،1:30 PM
عناصر قائمة أبل	-	مجلد	-	Wed, Feb 19, 1992 ،1:25 PM
لوحات التحكم	-	مجلد	-	Wed, Feb 19, 1992 ،1:25 PM
ملحقات النظام	-	مجلد	-	Wed, Oct 28, 1992 ،8:26 AM
ملف دفتر المسودة	29K	ملف	-	Fri, Oct 11, 1991 ،12:33 AM

The Text Utilities include a number of routines for converting and formatting date and time strings on the Macintosh. These routines allow you to specify each element of the date and time formats, including the number of digits used for each numeric element (for example, 3/01/90 or 3/1/90), the names of the months and the days of the week, and other characteristics such as the order of the elements and the use of a.m. and p.m. instead of a 24-hour clock.

Be careful about abbreviating the names of the weekdays (for instance, in English *S, M, T, W, Th, F,* and *S*). In Hebrew, for example, the names all begin with the same character, so the English convention would not be useful. Use instead the Text Utilities routines that give you the abbreviated versions provided by each script system.

Multiple calendars may be available on some Macintosh systems. The time-formatting and date-formatting routines in the Text Utilities are generalized enough that they can handle other calendar systems. The Gregorian calendar is the standard Macintosh calendar that is used in most of the world, but other calendars are also supported. See the description of the long-date-format resource in the appendix "International Resources" in this book for a list of defined calendar types.

Numbers and Symbols

Western numerals (*1, 2, 3,* and so forth) are not universal, and the decimal separator is not always the period. The formats of numbers vary widely. The Japanese writing system, for instance, uses the standard ASCII Western digits, 2-byte encodings of the same Western digits, and 2-byte Japanese number characters in two forms.

To accommodate differences in number and currency formats around the world, the Text Utilities provide routines that separate the presentation of numeric values from their internal representation. They allow a script system or your application to define separately how positive numbers, negative numbers, and zero values are presented. They allow you to specify what separators, digits, text annotations, marks (such as +), and literals (such as brackets or parentheses) can appear in numbers, and what kinds of padding can be used. In addition, they allow you to define how to represent positive and negative exponents for scientific notation. Each script system's numeric-format (`'itl0'`) and tokens (`'itl4'`) resources contain information used for formatting numbers.

Currency formats include the specification of the currency symbol (for example, $, £, or DM) and whether it precedes or follows the value. Each script system's numeric-format resource specifies formats for currency.

Use the regional forms of symbols such as the bullet (center dot, •). Tokens that allow you to define these symbols in a language-independent fashion are found in each script system's international resources; use the Script Manager to gain access to those tokens.

Note
Units of measure should be appropriate for the region you are targeting. For example, *lines per inch* is meaningless in the metric world. Units of measurement can be specified as metric or imperial (inches and miles). Each script system's numeric-format resource indicates the preferred measurement unit. You can use the Operating System Utilities function `IsMetric` to determine the appropriate unit of measure for the current script system. See *Inside Macintosh: Operating System Utilities.* ◆

Analyzing Characters

Analyzing characters is another common type of text-manipulation task. The Script Manager provides functions that let you analyze the size and type of individual characters. For example, with script systems that use 2-byte characters, you may need to determine what part of a character a single byte represents. In either 1-byte or 2-byte systems, you may need to know what type of character a particular character code represents. Character-type information is contained in a script system's string-manipulation (`'itl2'`) or encoding/rendering (`'itl5'`) resource.

For example, when searching for a single 1-byte character in text that may contain 2-byte characters, it is important not to mistake part of a 2-byte character for the character you are seeking. You can also determine whether a particular character is a letter, number, or punctuation mark, or whether or not it is uppercase. This information can be useful, for example, to filter input into specialized text fields. Also, for example, because several uppercase letters in the Cyrillic and Roman script systems are identical in appearance, you can detect an unwanted mixture of Cyrillic and Roman characters.

The Text Utilities provide a function that locates sequences of Roman characters (or characters of any other subscript) within non-Roman text. Use this routine when you want to separate out Roman characters into their own style runs, so that they can be formatted independently of the surrounding non-Roman text.

Searching, Modifying, and Converting Text

The Text Utilities provide several script-aware routines that you can use to modify the contents of strings or convert text from one form to another. You can use these routines on strings of any script system; the script-specific information they need is in the script's string-manipulation (`'itl2'`), tokens (`'itl4'`), encoding/rendering (`'itl5'`), or transliteration (`'trsl'`) resource.

For modifying strings, there are routines to

- convert case and strip diacritical marks from characters (such as for sorting)
- truncate a string to make it fit into a specified area on the screen
- search for a character sequence in a string and replace it with a different sequence (accounting for both 1-byte and 2-byte characters)

When searching, note that the text of some script systems can have accents or other diacritical marks that are considered optional. In Hebrew, for example, you may want to give the user the option to have search procedures ignore vowel and cantillation marks, because they are infrequently used in everyday writing. Note, however, that your application would have to provide this capability on its own; the Text Utilities stripping routines do not strip vowel or cantillation marks.

Different script systems have their own rules for dictionaries and hyphenation references. In searching text, your routines must be able to ignore text from script systems other than those to which the dictionaries and hyphenation references apply. As usual, it is your application's responsibility to track the script system of the text you manipulate; the script management system does not.

If you need to truncate a string, use the regional form of the ellipsis to indicate the truncation; different symbols may be expected in different languages. The Script Manager and the Text Utilities have routines that help you truncate strings and insert the proper symbol for an ellipsis.

Macintosh Human Interface Guidelines has guidelines for implementing intelligent cut-and-paste in your application. If the user cuts an entire word and pastes it in another location or document, you should make sure that the pasted word has proper word delimiters at its new location, and that extra word delimiters are not left at the location it was cut from. Applying intelligent cut-and-paste across all script systems requires complete understanding of word delimiters for each one. The Macintosh script management system does not provide support for this. However, the guidelines presented in *Macintosh Human Interface Guidelines* can work for any script system that uses spaces as word delimiters, and each script system sets a flag that you can access to determine whether it uses spaces. See the descriptions of script-variable selectors in the chapter “Script Manager” in this book.

Compilers, assemblers, and scripting-language interpreters usually parse sequences of characters to **tokens,** abstract entities that stand for variables, symbols, and quoted literals. Each script system provides tokenizing information in its tokens resource, for use by the Script Manager. Using the Script Manager you can create tokens recognizable by a parser in any script system.

The Script Manager also provides support for **transliteration,** the automatic conversion of text from one phonetic form or subscript to another within a single script system. In the Roman script system, this simply means case conversion. In Japanese, Chinese, and Korean script systems, it means the phonetic conversion of characters from one subscript to another. Script-specific information for transliteration is in a script’s string-manipulation or transliteration resource. With the Script Manager you can convert, for example, from Hiragana to Romaji and Romaji to Katakana in Japanese; from Bopomofo to Roman in Chinese; and from Roman to Jamo, Jamo to Hangul, Hangul to Jamo, and Jamo to Roman in Korean.

Finding Word Boundaries and Line Breaks

Finding word boundaries for word-selection and for line-breaking is a common, though often difficult, text-manipulation task. Word-selection methods differ among script systems. For example, the Thai script does not use spaces between words; the Thai system must detect word boundaries by parsing. The Text Utilities provide a procedure that you can use to determine word boundaries, in order to support double-clicking, highlighting of search targets, and so on. You can also use the same procedure to find word boundaries for line breaking; see “Text Layout” on page 1-71. The procedure works for all script systems and uses information in the script’s string-manipulation (`'itl2'`) resource.

Script Systems in Use

When a version of Macintosh system software is created for a particular country or region, its system script supports the writing system of that country or region. In addition, the system software's text strings are usually translated, and its icons and other graphical elements may be altered to fit the cultural conventions of the region. This process of adapting software to local use is called *localization.*

Localization of system software is performed by Apple Computer, Inc. In constructing a localized system, many different combinations of script capability and text translation are possible. For example, one localized version of Hebrew system software might use Hebrew text strings and Israeli currency, date, and calendar formats. Another might leave all text strings in English and use Roman formatting. In both, of course, the system script would be Hebrew. In another example, localized system software for India might possibly use Gurmukhi (an Indic script system) as the system script but leave all text strings in English, using the low-ASCII characters in the Gurmukhi character set.

This final section of the chapter discusses how a localized system is presented to a user. It shows the locations of the files and resources that make up the system script and any auxiliary scripts. It then describes how you can modify existing script systems or make additional auxiliary script systems available to the user. The section then summarizes how the user can switch among the available script systems. Finally, it shows how the user can alter the configurations of the script systems on the computer, including possibly selecting as script-system defaults resources that you provide.

More information on localization and localized versions of system software can be found in *Guide to Macintosh Software Localization.*

Installing and Enabling Script Systems

A user receives a script system in one of two forms: as a system script, already installed in the user's System file and System Folder; or as a secondary script consisting of a set of files that, if not present in the System file already, need to be installed before they can be used.

Initialization

The Operating System initializes the Script Manager at startup, and the Script Manager, along with WorldScript I and WorldScript II, initializes all installed script systems. If a script system is properly installed and successfully initialized, it becomes **enabled** (made available for use by the Script Manager and applications). For more information, see the discussion on testing for the Script Manager and script systems in the chapter "Script Manager." ◆

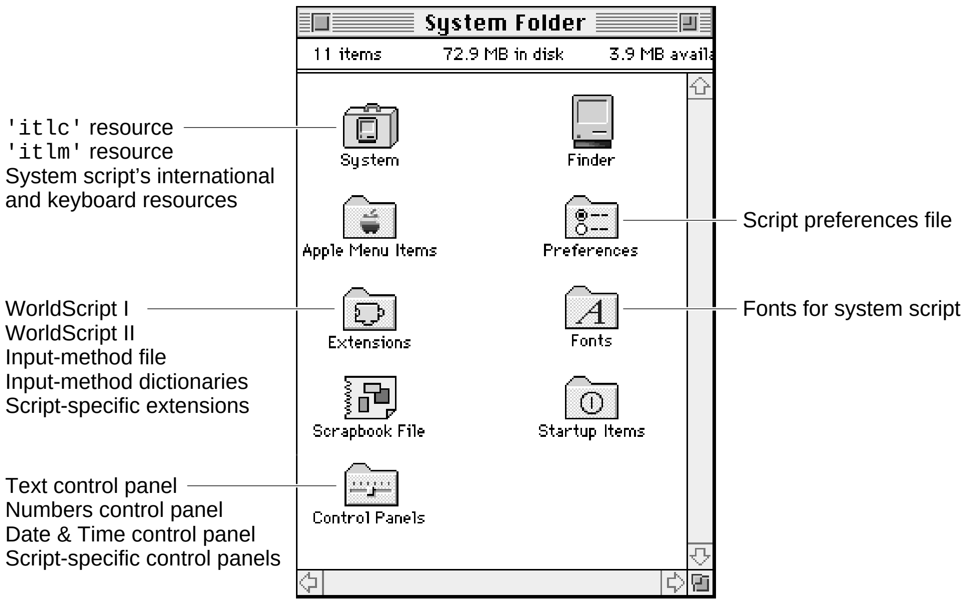
Components of the System Script

Because localization of system software involves more than installing script-system resources—for example, system and Finder text strings need to be translated—the user typically does not install a system script. However, if the user has two separate systems with two different localized versions of system software, the user can change system scripts by using the "Update Install" command in the installer to completely replace one localized version's system script (and all other localized resources) with those of the other localized version.

Once installed, the system script and associated files and resources are organized in the System Folder as follows (see Figure 1-59):

- The essential resources that make up the system script are in the System file. This includes the script's `'itlb'` resource and any of the following resources specified by the `'itlb'` resource: `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, `'itl5'`, `'trsl'`, `'itlk'`, `'KCHR'`, `'kcs#'`, `'kcs4'`, and `'kcs8'`.
- The System file also contains an international configuration resource (`'itlc'`) and a script-sorting resource (`'itlm'`).
- The Keyboard resources needed for each type of supported keyboard (`'KMAP'` and `'KCAP'`), though not considered part of any script system, are in the System file.
- If the system script is a 1-byte complex script system or a 2-byte script system, the Extensions folder contains a script extension: either WorldScript I or WorldScript II, respectively.
- If the system script is a 2-byte script system, the Extensions folder contains one or more input-method files. The Extensions folder may also contain one or more dictionary files needed by the input method.
- Depending on its individual needs or version, the system script may also have an extension file of its own, a file of type `'scri'` in the Extensions folder.
- The Fonts folder contains the fonts needed by the system script.
- If the system script provides a control panel for the user, its control panel file is in the Control Panels folder. If the control panel allows the user to save script settings, there is a script preferences file in the Preferences folder to hold those settings. (The file is created the first time the user changes any settings.) Note that this control panel and preferences file are separate from the Text, Numbers, and Date & Time control panels described under "User Control of Script Settings" beginning on page 1-107.
- If the system script needs additional files, they are in the System Folder.

Figure 1-59 System-script components in the System Folder



Components of Auxiliary Scripts

Auxiliary scripts consist of a set of resources and files mostly similar to those of a system script. The essential resources that make up the auxiliary script—an international bundle resource and any other international and keyboard resources needed by the script—may have been installed in the System file during system localization or may be contained in a file that is shipped separately from system software or applications. Other files are parallel to the files associated with a system script, as shown in Figure 1-59. (The closer a script system is to the U.S. version of the Roman script system, the fewer resources and files it has.)

To install a separately shipped secondary script from the Finder, the user can simply drag the contents of a folder containing the script's resources and files to the System Folder. The Finder automatically installs the files and resources properly, as follows:

- The Finder installs the resources from the script file into the System file. That includes the script's `'itlb'` resource and any of the following resources specified by the `'itlb'` resource: `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, `'itl5'`, `'trsl'`, `'itlk'`, `'KCHR'`, `'kcs#'`, `'kcs4'`, and `'kcs8'`.
- The Finder places all system extension files, including input-method files, dictionary files, and files of type `'scri'`, into the Extensions folder. This includes the WorldScript I or WorldScript II script extensions, if included.

- The Finder places all fonts for the script system into the Fonts folder.
- The Finder places any control panel documents for the script system into the Control Panels folder. (Once the user saves any new settings, a script preferences file is created in the Preferences folder.)
- The Finder places all other files into the System Folder.

If a script system has been installed but not yet enabled (if the computer has not been restarted), the user can take the script system's resources back out of the System file. (When the System file is opened, the Finder displays any script files that can be moved out of the System file.) Once the script system has been enabled, its resources can no longer be removed from the System file with the Finder.

Disabling script systems at startup

Holding down the Option–Space bar key combination at startup disables all (non-Roman) auxiliary scripts. This allows the user to remove auxiliary scripts from the System file that would normally have been enabled and thus impossible to remove from the Finder.

Holding down the Shift key at startup prevents system extension files from executing—including WorldScript I and WorldScript II. If the system script requires a script extension, system messages may not display properly. ◆

Apart from installing a script system itself, users can always move fonts into and out of the Fonts folder, and input methods into and out of the Extensions folder.

Installing Modifications to a Script System

Applications that are written to take advantage of the Macintosh script management system function correctly regardless of the localized version of Macintosh system software under which they run. However, it is also possible to tailor an individual application for a specific script system or set of scripts, or for a specific regional variation of the system script or other script.

To do so may require installing a new script system or a modified set of resources to replace those of a currently installed script system. (This is especially true if your target region is not already supported by a localized version of system software.) Either way usually involves modifying the System file. The Apple Computer system software licensing policy forbids shipping a modified System file, so you cannot install your replacement resources in a System file and ship it with your application. However, there are three other approaches you can take:

- If you create individual modified versions of an installed script system's resources—in order to implement region-specific sorting or formatting conventions—you can attach those resources to your application and have them replace the existing script system's resources whenever your application is running.

This method requires no modifications to the System file at all. For specific instructions, see the discussion on replacing a script system's default international resources in the chapter "Script Manager" in this book.

- If you want individual resources permanently installed in the System file, you can have the user run the Installer to install your resources. Contact Macintosh Developer Technical Support for information on how to use the Installer. The user will then be able to select or deselect your resources as defaults through the Text, Numbers, and Date & Time control panels. See "User Control of Script Settings" beginning on page 1-107.
- If you want to provide a complete script system with your application, you can ship it as a separate file in a folder along with fonts and any other assocated files. The user can then install it as an auxiliary script as described earlier, under "Components of Auxiliary Scripts."

  Your script system must be complete or it will not be enabled at startup. What constitutes a complete script system is described under testing for script systems in the chapter "Script Manager" in this book. The formats of the resources you need to include are described in the appendixes "International Resources" and "Keyboard Resources" in this book.

In general, it is not feasible to replace a system script, except by doing an "Update Install" from another complete localized system, as described on page 1-101. Although the user can replace individual resources in the System file by using a resource editor such as ResEdit, it is not possible to directly replace a system script with an auxiliary script because a system script requires an international configuration (`'itlc'`) resource, which is not part of any auxiliary script. Furthermore, replacing the system script is not the same as localizing all of the system software. A system script should support the system software it is shipped with, meaning that the language and icons of system menus, dialog boxes, and messages should reflect the system script. Merely replacing the system script does not accomplish that.

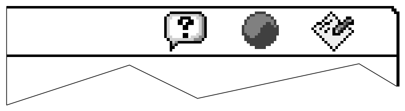
How the User Switches Among Script Systems

The script system for display of text is controlled by the application or by the system, based on which graphics port is active, which font is the current font, and what the states of the font force flag and international resources selection flag are.

The script system for text input, the keyboard script, is controlled by the user, either explicitly through a menu selection, or implicitly through choosing a font or selecting or clicking in displayed text of a particular script system. This section summarizes how the keyboard script is selected; for more complete information, see the discussions of keyboard settings and synchronization in the chapter "Script Manager" in this book.

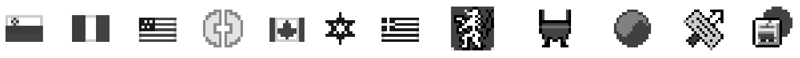
In any localized version of system software in which more than one script system is present, a small icon called a **keyboard icon** appears on the right side of the menu bar. Figure 1-60 shows the keyboard icon for the Korean script system, to the left of the application icon and to the right of the Help menu icon.

Figure 1-60 Menu bar with keyboard icon



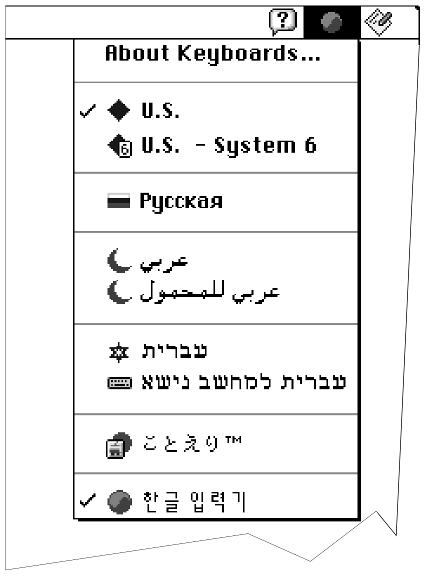
This symbol indicates which keyboard script, as represented by a keyboard layout or input method, is currently being used for text input. For example, the Arabic keyboard is represented by a crescent, the Hebrew keyboard by a Star of David, and common European keyboards by flags or other appropriate symbols. The Japanese input method is represented by an Apple icon in front of a rising sun; Chinese by a coin (Simplified) or a pot called a *Ding* (Traditional); Korean by the circular yin-yang symbol. The default Roman keyboard is represented by a blue diamond, except on versions of system software localized for the United States, in which it is represented by a U.S. flag. Figure 1-61 gives some examples of keyboard icons and input-method icons. Color Plate 4 shows a larger set of keyboard icons in color.

Figure 1-61 Keyboard icons and input-method icons



The keyboard icon serves as the title for the Keyboard menu; the user can click the keyboard icon to pull down the Keyboard menu. The Keyboard menu shows all keyboard layouts and input methods for all available keyboard scripts. The user makes a selection from the Keyboard menu in order to change the keyboard script, or to select among different keyboard layouts or input methods within a given script. See Figure 1-62.

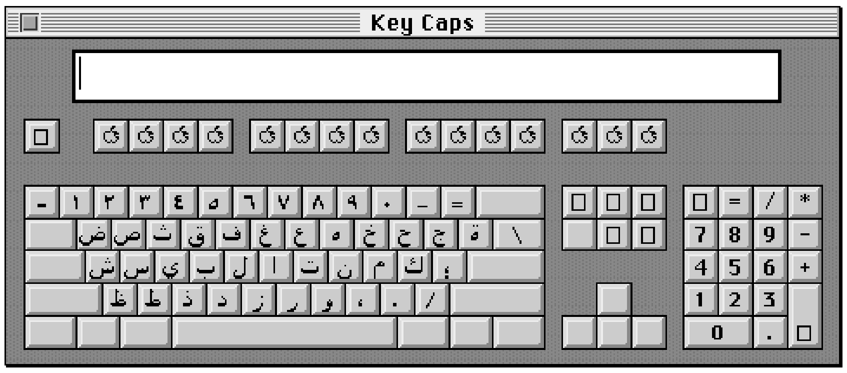
Figure 1-62 Keyboard menu



The Operating System provides keyboard equivalents for switching among script systems. In system software localized for the U.S., for example, if the user presses Command–Space bar, the Operating System switches the keyboard script to the “next” script system, meaning the default keyboard layout or input method for the next script system listed (down) the Keyboard menu. If the user presses Command–Option–Space bar, the Operating System switches to the next keyboard layout or input method within the current script system.

To see how the current keyboard layout functions, the user can select the Key Caps desk accessory. Whenever the keyboard script or keyboard layout changes, the Key Caps display changes to reflect the new character set and its arrangement on the keyboard. See Figure 1-63.

Figure 1-63 Arabic Key Caps



Application-Controlled Switching

Your application must synchronize the current font with the keyboard script whenever you are displaying characters as the user enters them. If the user changes fonts, you need to automatically change the keyboard script to correspond to the new font. Conversely, if the user changes keyboard scripts, you need to change the font appropriately before displaying the next character typed. Failure to do so can lead to incorrect text display. See "Font and Keyboard Synchronization" on page 1-90. ◆

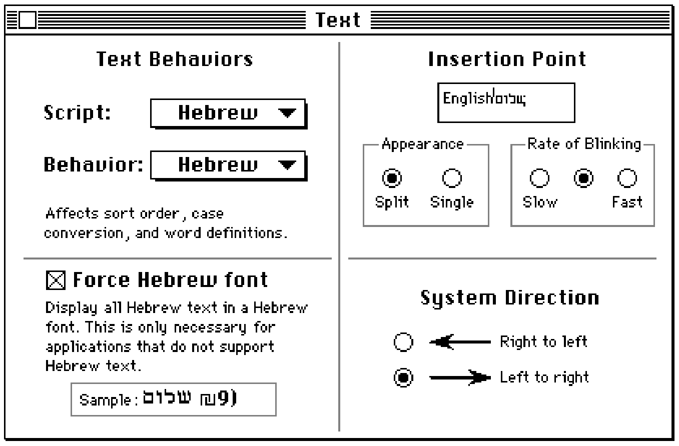
User Control of Script Settings

The script management system provides three control panels that allow the user to change the settings of certain script-system features and to save the settings across system restarts.

The **Text control panel**, shown in Figure 1-64, is available on non-U.S. versions of system software. It allows users to set the text behavior of any enabled script system, and may allow the user to set the system direction, the state of the font force flag, the caret style, and the rate of caret blinking. (Some of the settings are not available unless certain script systems are present.)

The appearance of the dialog box varies with the version of localized system software; Figure 1-64 represents a Text control panel for Hebrew system software localized to have all text strings in English.

Figure 1-64 Text control panel



Text

Text Behaviors

Script: Hebrew

Behavior: Hebrew

Affects sort order, case conversion, and word definitions.

☒ Force Hebrew font

Display all Hebrew text in a Hebrew font. This is only necessary for applications that do not support Hebrew text.

Sample : שלום

Insertion Point

Englishעברית

Appearance: [✓ Split] Single

Rate of Blinking: Slow [✓] Fast

System Direction

Right to left

[✓ Left to right]

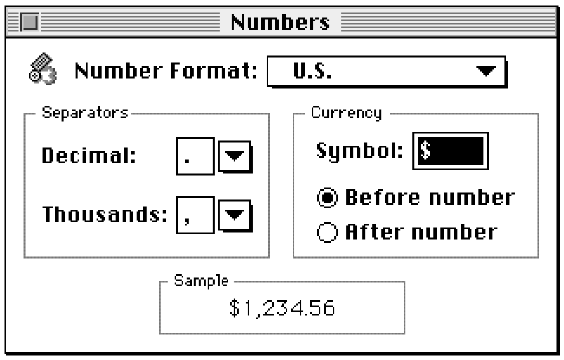
- The Text Behaviors settings control which string-manipulation (`'itl2'`) resource is used for sorting, case conversion, and word selection for the selected script system (including the system script). The choices are limited to the installed string-manipulation resources for the enabled script systems (including the Roman string-manipulation resource, which is always present). If more than one choice is available and the user changes this setting, the new setting is saved in the `itlbSort` field of the script's international bundle resource.
- The System Direction setting controls the primary line direction and alignment for all text and interface elements controlled by the system. The system direction may be set to either left to right or right to left. The user's selection is immediately reflected in the alignment of elements in all system and Finder dialog boxes and in all menus. It changes the setting of the system global variable `SysDirection`. The setting is also saved in the `itlcSysFlags` field of the system's international configuration resource. (This control appears only if at least one bidirectional script system is enabled.)
- The font force flag may be set to either `TRUE` or `FALSE`, which affects the setting of the Script Manager variable accessed through the `smFontForce` selector for the `GetScriptManagerVariable` function. The font force flag allows display of non-Roman text in an application that normally supports Roman text only. See the chapter "Script Manager" in this book. The setting made by the user is saved in the `itlcFontForce` field of the system's international configuration resource. (This control appears only if the system script supports font forcing.)

- The Insertion Point setting sets the caret style. The caret may appear either as a single caret or as a dual (split) caret (see Figure 1-46 on page 1-78 for an example). The setting made by the user is reflected in the value of the Script Manager general flags, accessed through the `smGenFlags` selector for the `GetScriptManagerVariable` function. See the chapter "Script Manager" in this book. The setting made by the user is saved in the `itlcFlags` field of the system's international configuration resource. (This control appears only if at least one bidirectional script system is installed.)

  The rate of caret blinking (slow, medium, or fast) affects the insertion point in text fields. The user's setting is saved in parameter RAM.

The **Numbers control panel,** shown in Figure 1-65, allows users to specify the basic number and currency formats for the system script. User settings made through this control panel are saved in the system script's numeric-format (`'itl0'`) resource.

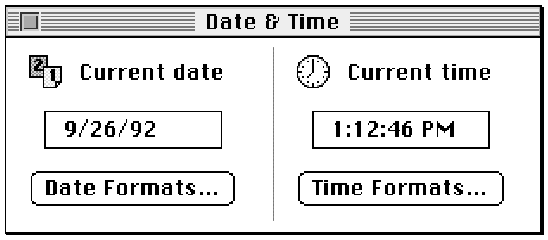
Figure 1-65 Numbers control panel



- The Number Format setting controls which numeric-format (`'itl0'`) resource is used for default number, currency, and short-date formats. The choices are limited to the installed numeric-format resources for the system script. If the user changes any of the default settings, a new setting called *custom* is created in the Number Format popup menu, and is saved as a new numeric-format resource for the system script; its ID is then saved in the `itlbNumber` field of the system script's international bundle resource.
- The Separators settings allow the user to override the default decimal separator and thousands separator for the system script. Suggested separators are presented in the popup menus for the settings, although the user can enter any 1-byte character for either separator. The settings made by the user are saved in the `decimalPt` and `thousSep` fields of the system script's custom numeric-format resource.
- The Currency settings allow the user to specify a currency symbol of up to three 1-byte characters or a single 2-byte character, and to choose whether the symbol precedes or follows a currency number. The settings made by the user are saved in the `currSym1` through `currSym3` fields of the system script's custom numeric-format resource.

The **Date & Time control panel,** shown in Figure 1-66, allows users to set the current date and time and to specify formatting preferences for both. The settings made with this control panel affect the display of dates and times by the system and Finder and by the Text Utilities date- and time-formatting routines, when the resources of the system script are used (that is, as long as the international resources selection flag is `TRUE`).

Figure 1-66 Date & Time control panel


Date & Time

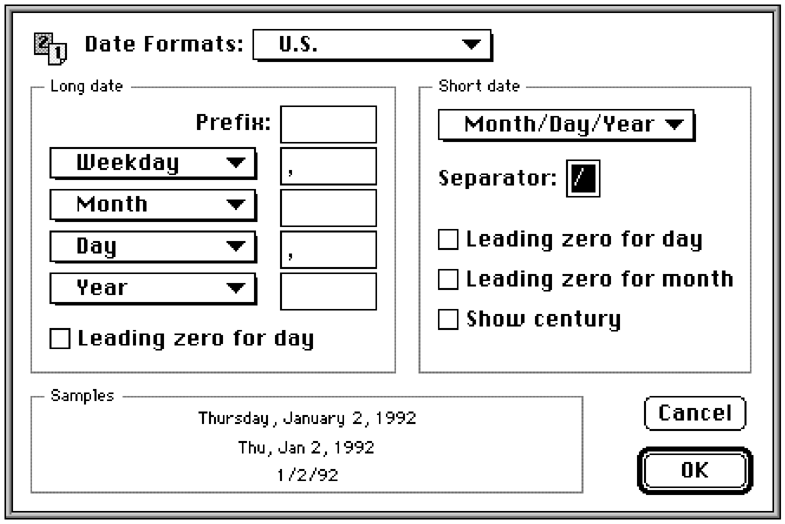
Current date: 9/26/92

Current time: 1:12:46 PM

Format settings are made with individual Date Formats and Time Formats dialog boxes. Custom user settings made through these dialog boxes are saved as new numeric-format (`'itl0'`) and long-date-format (`'itl1'`) resources for the system script.

The Date Formats dialog box sets date formats, as shown in Figure 1-67.

Figure 1-67 Date Formats dialog box (from Date & Time control panel)


Date Formats: U.S.

Long date

Prefix:

Weekday ,

Month

Day ,

Year

☐ Leading zero for day

Short date

Month/Day/Year

Separator: /

☐ Leading zero for day

☐ Leading zero for month

☐ Show century

Samples

Thursday, January 2, 1992

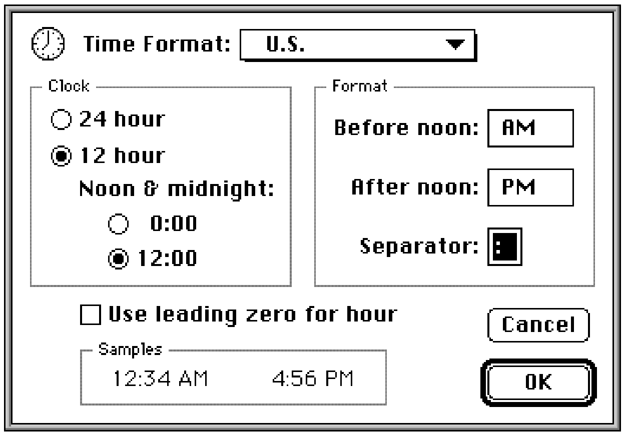
Thu, Jan 2, 1992

1/2/92

- The Date Formats setting allows the user to select a long-date-format (`'itl1'`) and numeric-format (`'itl0'`) resource to be used for date formatting. The choices are limited to the installed pairs of numeric-format and long-date-format resources for the system script. If the user changes any of the default settings, a new setting called "custom" is created in the Date Formats popup menu, and is saved as a new pair of numeric-format and long-date-format resources for the system script; their IDs are then saved in the `itlbNumber` field and `itlbDate` field of the system script's international bundle resource.
- The Long date settings allow the user to select what elements to include in a long date, what order they should be in, and what separators should be between them. The Long date settings also allow the user to specify the use of a leading zero for the day number in a long date. The settings made by the user are saved in the `days`, `months`, `suppressDay`, `lngDateFmt`, `st0` through `st4`, and `dayLeading0` fields of the system script's custom long-date-format resource.
- The Short date settings allow the user to select the order of date elements in a short date, and to specify a single (1-byte) character as separator. The Short date settings also allow the user to specify whether to use a leading zero for day number or month number, and whether to show the century. The settings made by the user are saved in the `dateOrder`, `dateSep`, and `shortDateFmt` fields of the system script's custom numeric-format resource.

The Time Formats dialog box sets time formats, as shown in Figure 1-68.

Figure 1-68 Time Formats dialog box (from Date & Time control panel)


Time Format: U.S.

Clock
- 24 hour
- [✓ 12 hour]
- Noon & midnight: 0:00 [✓ 12:00]

Format
- Before noon: AM
- After noon: PM
- Separator: :

☐ Use leading zero for hour

Samples: 12:34 AM 4:56 PM

- The Time Format setting allows the user to select a numeric-format (`'itl0'`) resource to be used for time formatting. The choices are limited to the installed numeric-format resources for the system script. If the user changes any of the default settings, a new setting called "custom" is created in the Time Formats popup menu, and is saved as a new pair of numeric-format and long-date-format resources for the system script; their IDs are then saved in the `itlbNumber` field and `itlbDate` field of the system srcipt's international bundle resource.

- The Clock settings allow the user to choose a 12- or 24-hour time cycle, and to specify whether midnight (and noon, if a 12-hour cycle) is considered to be hour 0 or hour 12. The settings made by the user are saved in the `timeCycle` field of the system script's custom numeric-format resource.
- The Format settings allow the user to specify a 1-byte character as separator for the time elements, and to specify morning and evening trailing strings (such as *AM* and *PM*) for the 12-hour cycle. The current separators and trailing strings are presented in the fields for the settings, but the user can enter any 1-byte character for the separator and any string of up to 4 bytes for either trailing string. The settings made by the user are saved in the `timeSep`, `mornStr`, and `eveStr` fields of the system script's custom numeric-format resource.

Script-specific control panels

In addition to the control panels described in this section, individual script systems may provide their own control panels for other purposes, such as allowing a user to select a custom calendar system, an associated font, or a set of numerals (ASCII or non-ASCII). The results of those selections may be kept in a script preferences file. ◆