This chapter describes the Script Manager, a core component of the Macintosh script management system. The Script Manager oversees script systems and gives you access to their features.

Read this chapter if you are writing a multiscript text-handling application and need access to the general settings and script-specific information provided by the Script Manager. Read this chapter also if you are writing a specialized application that parses source code or converts text among subscripts. Read this chapter also if you wish to modify the features or functions of an individual script system.

Before reading this chapter, you should be familiar with the Macintosh script management system, as described in the chapter "Introduction to Text on the Macintosh" in this book. Useful related information is found in the appendixes "International Resources," "Keyboard Resources," and "Built-in Script Support."

This chapter—and this book—do not catalog the features of individual script systems. More detailed information on the world's writing systems and the Macintosh script systems developed to support them can be found in *Guide to Macintosh Software Localization*.

The chapter gives a brief introduction to the Script Manager, and then shows how you can use the Script Manager to

- control default settings for text handling
- obtain information about a script system
- convert text through tokenization or transliteration
- modify a script system by replacing its resources or—in some cases—its routines

About the Script Manager

The Script Manager is at the center of the Macintosh script management system. It makes script systems available. It coordinates the interaction between many parts of Macintosh system software and those available script systems.

The Script Manager also provides several services directly to your application. Through them you can get information about the current text environment, modify that environment, and perform a variety of text-handling tasks.

The Script Manager has evolved through several versions. It started with sole responsibility for all international-compatibility and multilanguage text issues, but as more power and features have been added, many of its specific functions have been moved to the other parts of system software.

The Script Manager and the Script Management System

The Script Manager manages script systems. It monitors their initialization and maintains variables and data structures that affect their functioning. It makes sure that all initialized script systems are complete in terms of having the required international resources and fonts. It gives applications as well as other parts of system software their principal access to script systems' features.

The Script Manager works closely with the other managers that make up the Macintosh script management system, in particular the Text Utilities and QuickDraw. The Text Utilities include many script-aware routines that manipulate text, and QuickDraw provides script-aware measuring and drawing routines for text. When your program or a system routine makes a script-aware Text Utilities or QuickDraw call, it commonly results in an internal call to the Script Manager, to access a global setting or the data of a script system.

TextEdit also relies on the Script Manager, both directly and through the Text Utilities and QuickDraw, to make sure that it handles text correctly in any script system. The Font Manager, the Text Services Manager, and the Dictionary Manager use information maintained and provided by the Script Manager.

Other components of Macintosh system software also interact with the Script Manager. The Finder uses the Script Manager to correctly input, display, and sort file and folder names across all localized versions of system software. The Menu Manager, the Event Manager, the Process Manager, the Operating System Utilities, and the Component Manager all work with the Script Manager, directly or indirectly, to obtain the information necessary to properly handle multiscript text.

The Script Manager and Applications

The Script Manager is your application's principal interface—either direct or indirect—with any of the script systems that may be available on the user's computer. For many text-related tasks, the Script Manager's role is transparent; when you make a script-aware Text Utilities or QuickDraw call while processing text, that routine may get the information it needs through the Script Manager. For example, when you call the QuickDraw procedure `DrawText` to draw a line of text, `DrawText` in turn calls the Script Manager to determine which script system your text belongs to before drawing it.

In other situations you may need to call the Script Manager explicitly, to properly interpret the text you are processing. Those situations are the principal subject of this chapter.

The Script Manager provides services that fall into four general categories: controlling settings, obtaining information, modifying text, and modifying script systems. Any text-handling application that you write, unless it relies solely on TextEdit, will need to use some of those services. Almost any text application, for example, needs to call the

`GetScriptManagerVariable` function. Other calls are for specialized programs only. The `IntlTokenize` function, for example, is only for specialized programs that parse highly structured text such as source code, mathematical expressions, or formatted numbers.

These are the services provided by the Script Manager in each of the four categories:

- Controlling settings. The routines in this category are of general interest and are used by most text applications. With these routines you can
  - check and set the system direction, a global variable that controls the default alignment of text and can affect the order in which blocks of mixed-directional text are drawn.
  - check and set Script Manager variables, private variables used by the Script Manager to keep track of information that is general to the text environment.
  - check and set script variables, private variables used by script systems to keep track of their own configurations.
  - make keyboard settings that affect text input, so that users can enter text in any script system and you can display it properly.
- Obtaining information. Many of the routines in this category are of general interest and are used by most text applications. With these routines you can
  - determine script codes from font information. Most applications need this information.
  - analyze characters for size (in bytes) and type. Applications that work with 2-byte script systems need size information, and many applications need character-type information.
  - directly access a script system's international resources. Most applications need this information only to pass it to other routines. Some applications also use it to inspect or modify individual tables or other data within a resource.
- Converting text. The routines in this category are used by specialized applications only. (Text-modification routines of general interest to applications are described in the chapter "Text Utilities" in this book.) With these routines you can
  - tokenize text: convert source text from any script system into script-independent tokens.
  - transliterate text: phonetically convert text from one subscript to another within a script system.
- Modifying script systems. The routines in this category are used for specialized purposes, such as providing regional variants to existing script systems or assigning script-specific features to individual documents or applications. With these routines you can
  - replace or modify the default international resources of a script system.
  - replace individual text-handling routines in certain script systems.

Evolution of the Script Manager

The Script Manager is only one of several system software managers that make up the Macintosh script management system, but its position is central.That central position stems from the fact that, in previous versions, the Script Manager alone (including the International Utilities) was responsible for all international text processing.

The first version of the Script Manager was released with Macintosh System 4.1. Table 6-1 shows the routines and some of the features of Script Manager 1.0, and the additional routines and features that have marked each successive version of the Script Manager (and International Utilities). Some of the added routines rendered earlier ones obsolete, whereas others brought new capabilities.

Table 6-1 Evolution of the Script Manager

Version	New routines, other additions and enhancements
1.0	`Char2Pixel`, `CharByte`, `CharType`, `DrawJust`, `FindWord`, `Font2Script`, `FontScript`, `GetAppFont`, `GetDefFontSize`, `GetEnvirons`, `GetMBarHeight`, `GetScript`, `GetSysFont`, `GetSysJust`, `HiliteText`, `IntlScript`, `KeyScript`, `MeasureJust`, `Pixel2Char`, `SetEnvirons`, `SetScript`, `SetSysJust`, `Transliterate` introduced. New international resources defined.
2.0	`FindScriptRun`, `Format2Str`, `FormatStr2X`, `FormatX2Str`, `GetFormatOrder`, `InitDateCache`, `IntlTokenize`, `IULDateString`, `IULTimeString`, `LongDate2Secs`, `LongSecs2Date`, `LwrString`, `LwrText`, `ParseTable`, `PortionText`, `ReadLocation`, `Str2Format`, `String2Date`, `String2Time`, `StyledLineBreak`, `ToggleDate`, `ValidDate`, `VisibleLength`, `WriteLocation` added.
2.17/ 2.21*	Enhanced `'itl2'` resource. Full support for Standard Roman character set. New token types defined.
7.0	`IUClearCache`, `IUGetItlTable`, `IULangOrder`, `IUScriptOrder`, `IUStringOrder`, `IUTextOrder`, `LowerText`, `NChar2Pixel`, `NDrawJust`, `NFindWord`, `NMeasureJust`, `NPixel2Char`, `NPortionText`, `StripText`, `StripUpperText`, `TruncString`, `TruncText`, `UpperText` added. Support for scaled justified text layout. Implicit script codes, new selectors. New keyboard resources, enhanced U.S. `'KCHR'` resource.
7.1	`CharacterByteType`, `CharacterType`, `FillParseTable`, `GetQDPatchAddress`, `GetScriptUtilityAddress`, `SetQDPatchAddress`, `SetScriptUtilityAddress`, `TransliterateText` added to Script Manager; several existing routines renamed. Many additional new and renamed routines moved to other managers such as Text Utilities and QuickDraw. WorldScript extensions created.

* In hexadecimal, 2.17 is $211, and 2.21 is $215. See Table 6-2 on page 6-9.

The most extensive changes, in terms of how the Script Manager is documented, have been the most recent. Many of the routines described throughout *Inside Macintosh: Text* are previous Script Manager routines that have been relocated and possibly enhanced or renamed. They were moved to be documented alongside routines of similar purpose in other managers. Many of the early Script Manager routines listed in Table 6-1 are obsolete and are no longer documented at all. See the appendix "Renamed and Relocated Text Routines" in this book for information on the current status and location of any previous Script Manager or International Utilities routines not found in this chapter.

Using the Script Manager

This section explains how you can use the Script Manager in performing four types of text-related tasks. Script Manager routines can help you with

- accessing and controlling the configuration of the text-handling environment, by
  - determining the version of the Script Manager and the number of active script systems
  - checking and setting the system direction
  - checking and modifying Script Manager variables
  - checking and modifying script variables
  - making keyboard settings that affect text input
- obtaining script-related information to help you process text, by
  - determining script codes from font information
  - using character-type information for searching and analyzing text
  - directly accessing a script system's international resources
  - using specific tables within a script system's international resources
- converting text for specialized purposes, by
  - converting source text from any script system into script-independent tokens
  - transliterating text from one subscript to another within a script system
- modifying the features of a script system, by
  - replacing or modifying the default international resources of a script system
  - replacing individual text-handling routines in 1-byte complex script systems

Testing for the Script Manager and Script Systems

This section describes how to use the `Gestalt` function to test for the current version of the Script Manager and the number of active script systems. For details on the `Gestalt` function, see the Gestalt Manager chapter of *Inside Macintosh: Operating System Utilities*.

The Operating System initializes the Script Manager at startup. The Script Manager then initializes the Roman script system. Next the Script Manager initializes any other installed 1-byte simple script system whose `smsfAutoInit` bit (see page 6-69) is set. The Script Manager then allows the script extensions WorldScript I and WorldScript II (if present) to initialize all installed 1-byte complex and 2-byte script systems.

When initializing a script system, the Script Manager or script extension first checks to make sure that there is enough memory for the script system, and then checks that an international bundle resource is present in the System file and that at least one font in the proper ID range for that script system is present in the System file or in the Fonts folder. If these resources are present, the script system is considered to be **enabled** (available for use by the Script Manager and applications). If the required resources are not available, the script system remains disabled.

Note
The Script Manager is fully loaded and all script systems are enabled before any files of type `'INIT'` in the Extensions folder are launched. Thus, all Script Manager routines can be called from system extensions. ◆

Use `Gestalt` with the `gestaltScriptMgrVersion` selector to obtain a result in the response parameter that identifies the version number of the Script Manager. This is the same value returned by a call to the `GetScriptManagerVariable` function with the selector constant `smVersion`. Table 6-1 on page 6-6 lists some of the routines and features available with the principal versions of the Script Manager.

Table 6-2 gives more detail on the version numbers returned by `Gestalt` or by `GetScriptManagerVariable` with the selector `smVersion`, for all versions of system software and all versions of the Script Manager. It also shows the Roman script system versions returned by the `GetScriptVariable` function with the selector `smScriptVersion`.

Table 6-2 Version numbers for the Script Manager and Roman script system

System software version	Script Manager (newer ROMs)	Script Manager (older ROMs)*	Roman script system
6.0.3 and earlier	N.A.	<= $20F	<= $101
6.0.4 Roman	$215	$211	$101
6.0.4 non-Roman	$216	$212	$101
6.0.5 all	$217 (= 2.23)	$213	$101
Above this line, minor version numbers are binary; below, they are BCD:			
6.0.7 all†	$231 (= 2.3.1)	$230	$101
J-6.0.7.1 (Japanese)‡	$231	$230	$101
6.0.8 all	$231	$230	$101
6.1 (non-Roman)	$241	$240	$101
7.0	$700	$700	$700
7.0.1 Roman	$700	$700	$700
7.0.1 non-Rman	$701	$701	$701
7.1	$710	$710	$710

* On Macintosh Plus, Macintosh SE, Macintosh II, Macintosh IIx, Macintosh IIcx, Macintosh SE/30, Macintosh Classic. Other CPUs have newer ROMs.
† `Gestalt` actually returns $606 as the system version for non-Roman versions of system 6.0.7.
‡ `Gestalt` actually returns $609 as the system version for system J-6.0.7.1.

Note
In versions of system software earlier than 6.0.7, the major and minor version numbers are each treated as if they were binary. Thus a result of $217 from `Gestalt` means a Script Manager version of 2.23 (in decimal). Starting with system 6.0.7, version numbers are returned as binary-coded decimal numbers, so a result of $710 means a Script Manager version of 7.10 (or 7.1.0). ◆

Use the `Gestalt` selector `gestaltScriptCount` to obtain a result in the response parameter that gives the number of active script systems. This is the same value returned by a call to the `GetScriptManagerVariable` function with the `smEnabled` selector.

Obtaining the number of active script systems is most useful for testing whether more than a single script system is present. If the result is 1, only the Roman script system is present and text-handling is simplest. If the result is greater than 1, at least one non-Roman script system is present, and your application needs to be able to handle its text.

Controlling Settings

The first principal use for the Script Manager is in controlling the settings that determine the current characteristics of the text-handling environment. The Script Manager gives you access to many variables, fields, flags, and files that affect how script systems function and how text is manipulated and displayed. The routines described in this section are of general interest and are used by most text applications. You can use these Script Manager routines to

- set the system direction
- access Script Manager variables
- access script variables
- determine the keyboard script, keyboard layout, and input method

Checking and Setting the System Direction

The **system direction** is a global setting that is commonly used to define the primary line direction for text display. The system direction is specified by the value of the global variable `SysDirection`. The value of `SysDirection` is 0 for a left-to-right primary line direction and –1 for a right-to-left primary line direction.

System direction always controls the alignment (right or left) of interface elements such as menu items and dialog box items that are drawn by the system. It can also affect caret placement and the order in which blocks of text are drawn or highlighted in bidirectional script runs and in multiscript lines.

QuickDraw, TextEdit, and other parts of system software that use TextEdit set the system direction before drawing text. Although applications can format and draw text independently of the current value of system direction, applications that follow suggested procedures for text layout typically set the system direction before laying out and drawing any text. See, for example, the description of the `GetFormatOrder` function in the chapter "QuickDraw Text" in this book.

The default value for `SysDirection` usually corresponds to the primary line direction of the system script; it is initialized from the system's international configuration (`'itlc'`) resource at startup. The user can change the system direction from the Text control panel if a bidirectional script system is present.

If your application uses `SetSysDirection` to change the system direction in order to correctly order script runs in a line of text while drawing, be sure to first call `GetSysDirection` to save the original value. Then call `SetSysDirection` again at the appropriate time—such as when your application becomes inactive—to restore `SysDirection` to its original value.

Checking and Setting Script Manager Variables

The `GetScriptManagerVariable` and `SetScriptManagerVariable` functions let you check and set the values of the Script Manager variables, general environmental settings that the Script Manager maintains for all script systems.

These functions give you access to a large variety of general script-related information, including whether one or more bidirectional script systems is present, whether one or more 2-byte script systems is present, and what the states of the font force and international resources selection flags are.

You specify the variable you want to access with a **selector,** an integer constant that controls the function of a multipurpose routine. You pass a selector as a parameter to `GetScriptManagerVariable` or `SetScriptManagerVariable`. (The variables themselves are private and you cannot access them directly.) Table 6-3 lists the selector constants and the Script Manager variables they affect. See "Selectors for Script Manager Variables" beginning on page 6-61 for complete explanations of the selectors and variables.

Table 6-3 Script Manager variables accessed through `GetScriptManagerVariable/SetScriptManagerVariable`

Selector constant	Explanation
`smVersion`	Script Manager version number
`smMunged`	Modification count
`smEnabled`	Script count (0 if Script Manager not enabled)
`smBidirect`	Bidirectional script present flag
`smFontForce`	Font force flag
`smIntlForce`	International resources selection flag
`smForced`	Script-forced result flag
`smDefault`	Script-defaulted result flag
`smPrint`	Print action vector
`smSysScript`	System script code
`smLastScript`	Previous keyboard script
`smKeyScript`	Current keyboard script
`smSysRef`	System Folder volume reference number
`smKeyCache`	(obsolete, not used)
`smKeySwap`	Handle to keyboard-swap (`'KSWP'`) resource
`smGenFlags`	Script Manager general flags

continued

Table 6-3 Script Manager variables accessed through `GetScriptManagerVariable/SetScriptManagerVariable` (continued)

Selector constant	Explanation
`smOverride`	Script override flags (reserved)
`smCharPortion`	Intercharacter/interword spacing proportion
`smDoubleByte`	2-byte script present flag
`smKCHRCache`	Pointer to current keyboard-layout (`'KCHR'`) data
`smRegionCode`	Region code for system script
`smKeyDisableState`	Current disable state for keyboards

The following code fragment shows how to use the `GetScriptManagerVariable` function to get the Script Manager version number. This is the same value as that returned by the `Gestalt` function using the `gestaltScriptMgrVersion` selector.

```
VAR
   selectorValue: LongInt;
BEGIN
   selectorValue := GetScriptManagerVariable(smVersion);
END;
```

The `SetScriptManagerVariable` function allows you to change many text-related settings, including

- the font force flag
- the international resources selection flag
- the current keyboard script
- the Script Manager general flags, which include control of the display of the keyboard icon and the dual caret in TextEdit
- the proportion of intercharacter versus interword spacing, when laying out lines of justified text (in non-Roman script systems)

Listing 6-1 shows how to use the `SetScriptManagerVariable` function to specify the display of a dual caret in mixed-directional text. You do this by setting the appropriate bit of the Script Manager general flags field after retrieving it with the `GetScriptManagerVariable` function.

Listing 6-1 Specifying a dual caret with `SetScriptManagerVariable`

```
FUNCTION MySetDualCaret: OSErr;
VAR
   myErr:         OSErr;
   selectorValue: LongInt;
   flagValue:     LongInt;
BEGIN
   flagValue := BitShift($0001,smfDualCaret);
   selectorValue := GetScriptManagerVariable(smGenFlags);
   selectorValue := BitOr(selectorValue, flagValue);
   myErr := SetScriptManagerVariable(smGenFlags, selectorValue);
   MySetDualCaret := myErr;
END;
```

You can also use `SetScriptManagerVariable` to change the settings of the font force flag and the international resources selection flag, two flags that affect which script systems are used for text display and date/time/number formatting, respectively. See "Determining Script Codes From Font Information" beginning on page 6-21.

If you are using `SetScriptManagerVariable` to change the value of a variable for a specific task, first call `GetScriptManagerVariable` to retrieve the variable's original value, and save that value. Then call `SetScriptManagerVariable` and perform your task. Finally, restore the original value of the Script Manager variable with another call to `SetScriptManagerVariable` as soon as possible, so that other applications or software components that use the Script Manager will find the values they expect.

Checking and Setting Script Variables

The `GetScriptVariable` and `SetScriptVariable` functions let you retrieve and set script variables, local variables maintained for each script system by the Script Manager.

These functions give you access to a large variety of script-specific information, including the primary line direction for the script system, the default alignment for text in the script system, the script system's preferred system font and size, and its preferred application font and size.

You specify the script system whose variables you want to access with an explicit script code, or with an implicit script code specifying the system script or the font script. You specify the variable you want to access with a selector constant passed as a parameter to `GetScriptVariable` or `SetScriptVariable`. Table 6-3 lists the selector constants and the script variables they affect. See "Selectors for Script Variables" beginning on page 6-65 for complete explanations of the selectors and variables.

Table 6-4 Script variables accessed through `GetScriptVariable/SetScriptVariable`

Selector constant	Explanation
`smScriptVersion`	Script-system version number
`smScriptMunged`	Modification count
`smScriptEnabled`	Script-enabled flag
`smScriptRight`	Right-to-left line direction flag
`smScriptJust`	Default alignment (left or right)
`smScriptRedraw`	Amount of line to redraw when changing a character
`smScriptSysFond`	Preferred system font
`smScriptAppFond`	Preferred application font
`smScriptNumber`	Numeric-format (`'itl0'`) resource ID
`smScriptDate`	Long-date-format (`'itl1'`) resource ID
`smScriptSort`	String-manipulation (`'itl2'`) resource ID
`smScriptFlags`	Script flags
`smScriptToken`	Tokens (`'itl4'`) resource ID
`smScriptEncoding`	Encoding/rendering (`'itl5'`) resource ID
`smScriptLang`	Language code for script
`smScriptNumDate`	Current numeral code and calendar code
`smScriptKeys`	Keyboard-layout (`'KCHR'`) resource ID
`smScriptIcon`	Keyboard icon family ID
`smScriptPrint`	Print action routine for script
`smScriptTrap`	Pointer to script record dispatch routine entry point (for internal use)
`smScriptCreator`	Creator name for script file
`smScriptFile`	Filename for script file

Table 6-4 Script variables accessed through `GetScriptVariable/SetScriptVariable` (continued)

Selector constant	Explanation
`smScriptName`	Name of script system
`smScriptMonoFondSize`	Preferred font and size for fixed-width font
`smScriptPrefFondSize`	(unused)
`smScriptSmallFondSize`	Preferred font family and size for small text
`smScriptSysFondSize`	Preferred system font family and size
`smScriptAppFondSize`	Preferred application font family and size
`smScriptHelpFondSize`	Preferred Balloon Help font family and size
`smScriptValidStyles`	Valid text styles for script
`smScriptAliasStyle`	Text styles to use for aliases

You can use the `GetScriptVariable` function to get, for example, the default application font family (`'FOND'`) ID and size. In the following code fragment, the application uses the constant `smSystemScript` to specify that it is the system script whose font ID is needed. The ID is returned in the high-order word and the size is returned in the low-order word. The application then sets the appropriate graphics port fields to those values.

```
VAR
   myAppFont: LongInt;
BEGIN
   myAppFont := GetScriptVariable(smSystemScript,
                                  smScriptAppFondSize);
   TextFont(HiWord(myAppFont));
   TextSize(LoWord(myAppFont));
END;
```

Listing 6-2 shows how to represent font names correctly using the proper script for that font. First you call the Font Manager `GetFNum` procedure to get the font family ID using the font name. You call the `FontToScript` function using that font family ID to get the value of the associated script code. You then call `GetScriptVariable` with the `smScriptSysFond` selector to determine the font family ID for the preferred system font for the specified script. Finally, you call the QuickDraw `TextFont` procedure with that font family ID to set the font ID of the current graphics port to the preferred system font of the specified script.

Note
The Menu Manager `AddResMenu` procedure automatically represents font names in their associated script for `'FOND'` resources. If you need to display font names elsewhere than in the Font menu (for instance, using the List Manager), be sure to use a technique such as that shown in Listing 6-2. ◆

Listing 6-2 Representing font names correctly in the script for that font

```
PROCEDURE MySetTextFont(fontName: Str255);
VAR
   scriptFont: LongInt;
   scriptNum:  Integer;
   theNum:     Integer;

BEGIN
                              {from font name, get font ID}
   GetFNum(fontName, theNum); {use font ID to get script code, }
                              { then get preferred system font ID}

   scriptNum := FontToScript(theNum);
   scriptFont := GetScriptVariable(scriptNum, smScriptSysFond);

                              {now set the current grafPort's }
   TextFont(scriptFont);      { font ID to that font}
END;
```

The `SetScriptVariable` function allows you to change many script-specific settings, including the default configuration settings for the script system, which are initialized from a script system's international bundle (`'itlb'`) resource. You call `SetScriptVariable` with the appropriate script constant and selector to indicate the setting you want changed. Listing 6-3 shows how to use the `SetScriptVariable` function to set the size of the Balloon Help font to the size passed in the parameter `theSize`:

Listing 6-3 Setting the size of the Balloon Help font

```
PROCEDURE MySetHelpFontSize(theSize: LongInt);
VAR
   myErr:         OSErr;
   myHelpFont:    LongInt;

BEGIN
   theSize := BitAnd(theSize, $0000FFFF);
```

```
                                        {keep low word only}
   myHelpFont := GetScriptVariable(smSystemScript,
                                   smScriptHelpFondSize);
   myHelpFont := BitAnd(myHelpFont, $FFFF0000);
                                        {keep high word only}
   myErr := SetScriptVariable(smSystemScript,
                              smScriptHelpFondSize,
                              BitOr(myHelpFont,theSize));
   IF myErr <> noErr THEN DoError(myErr);
END;
```

If you are using `SetScriptVariable` to change the value of a variable for a specific task, first call `GetScriptVariable` to retrieve the variable's original value, and save it. Then call `SetScriptVariable` and perform your task. Finally, restore the original value of the script variable with another call to `SetScriptVariable` as soon as possible, so that other applications or software components using that script system will find the values they expect.

Making Keyboard Settings

The Script Manager `KeyScript` procedure lets you control the script system, keyboard layout, and input method used for text input. It also lets you make other settings related to text input.

You use the `KeyScript` procedure to change the keyboard script, the script system that controls text input. You also use it to switch among different keyboard layouts, resources that define the character sets and key positions for text input in a script system. You can also use it to switch among input methods, software facilities that allow text input in 2-byte script systems. If your application supports multiple languages, use `KeyScript` to change the keyboard script when the user changes the current font. For example, if the user selects Geezah as the current font or clicks the cursor within a run of text that uses the Geezah font, your application needs to set the keyboard script to Arabic. To do this, use the `FontToScript` function to find the script for the font, then use `KeyScript` to set the keyboard.

In addition, your application can check the keyboard script (using the `GetScriptManagerVariable` function) in its main event loop; if the keyboard script has changed, you can set the current font to the last-used font, application font, or system font of the new keyboard script (determined by a call to the `GetScriptVariable` function). This action saves the user from having to set the font manually after changing the keyboard script.

The system software performs the equivalent of calling `KeyScript` in response to the user selecting a keyboard layout or input method from the Keyboard menu. It also does the same when the user types Command–Option–Space bar (to select the next keyboard layout or input method within the same script system), or Command–Space bar (to select the next script system in the Keyboard menu).

When you call `KeyScript`, you pass it a `code` parameter that can explicitly specify a keyboard script by script code, or can implicitly specify a keyboard script, keyboard layout, input method, or other setting. Values for `code` equal to or greater than zero are interpreted as normal script codes. Several negative codes specify switching among keyboard scripts, keyboard layout, or input methods. Others toggle line direction or input method and are available only with certain script systems. Still others disable or enable keyboard layouts or keyboard scripts. Table 6-5 lists the valid constants for the `code` parameter.

Table 6-5 Constants for the `code` parameter in the `KeyScript` procedure

Constant	Value	Expanation
(any script code)	0…64	Switch to specified script
`smKeyNextScript`	–1	Switch to next script in Keyboard menu
`smKeySysScript`	–2	Switch to the system script
`smKeySwapScript`	–3	Switch to previously used script
`smKeyNextKybd`	–4	Switch to next keyboard layout or input method in Keyboard menu (within current script)
`smKeySwapKybd`	–5	(not implemented)
`smKeyDisableKybds`	–6	Disable keyboard layouts not in system script or Roman script
`smKeyEnableKybds`	–7	Enable keyboard layouts for all enabled scripts
`smKeyToggleInline`	–8	Toggle inline input for current script (available if 2-byte script present)
`smKeyToggleDirection`	–9	Toggle default line direction (available if bidirectional script present)
`smKeyNextInputMethod`	–10	(not implemented)
`smKeySwapInputMethod`	–11	(not implemented)
`smKeyDisableKybdSwitch`	–12	Disable switching out of current keyboard layout
`smKeySetDirLeftRight`	–15	Set primary line direction to left-to-right (available if bidirectional script present)
`smKeySetDirRightLeft`	–16	Set primary line direction to right-to-left (available if bidirectional script present)
`smKeyRoman`	–17	Set keyboard script to Roman (available only if multiple scripts present)

The `smKeyDisableKybds` selector is available for your use, although it is primarily used by the Finder or other parts of the system under special circumstances. For example, when the user enters the name of a file in a Standard-File dialog box, text input must be restricted to scripts that display correctly in the Finder and in dialog boxes, menus, and alert boxes. In that situation the system software calls `KeyScript` with the `smKeyDisableKybds` selector to disable keyboard input temporarily in any script system except Roman or the system script. Keyboards in other script systems then appear disabled in the Keyboard menu. When the user completes the filename entry, the system calls `KeyScript` again with a selector of `smKeyEnableKybds` to reenable keyboard input in all enabled script systems.

The `smKeyDisableKybdSwitch` selector is also available for your use, although it is primarily used by the Finder. When keyboard layouts and script systems are being moved into or out of the System file by the user, changing the current keyboard or keyboard script may corrupt files or cause other unpredictable results. To prevent all keyboard switching and to disable all the Keyboard menu items, the Finder calls `KeyScript` with the selector `smKeyDisableKybdSwitch`. When the move has been completed, the Finder again calls `KeyScript` with a selector of `smKeyEnableKybds` to reenable keyboard switching.

If you call `KeyScript` with `code` = `smKeyRoman` on a system in which only the Roman script system is enabled, nothing happens. However, if you call `KeyScript` with `code` = 0 (to select the Roman script system), it forces an update that selects the current default Roman keyboard layout.

IMPORTANT

Although it is possible to change the keyboard script without changing the keyboard layout—by calling the `SetScriptManagerVariable` function with the `smKeyScript` selector—it violates the user interface paradigm and creates problems for other script management routines. ▲

Synchronizing the Font Script and Keyboard Script

To keep the user from accidentally entering meaningless characters, you must always keep the keyboard script synchronized with the font script, so that the glyphs displayed on the screen match the characters entered at the keyboard. You can synchronize the scripts in two ways: by setting the keyboard script when the font script changes, and by setting the font script when the keyboard script changes.

Setting the Keyboard Script From the Font Script

Set the keyboard script from the font script when the user selects a new font or when the user clicks in or selects text.

- If the user selects a new font from the Font menu, call `TextFont` to set the current font to that font. Then set the keyboard script to the script system of that font.
- If the user clicks in or selects a text area, set the current font to be the font, size, and style of the text where the click occurred. Then set the keyboard script to the script system of that font.

Listing 6-4 is an example of code to use for setting the keyboard script from the font script. Once you have obtained the script code value from the font family ID using the `FontToScript` function (see "Determining Script Codes From Font Information" beginning on page 6-21), you call the `GetScriptManagerVariable` function with the `smKeyScript` selector to determine the keyboard script. If the font script and the keyboard script are not the same, call the `KeyScript` procedure to change the keyboard script.

Listing 6-4 Setting the keyboard script from the font script

```
PROCEDURE MySetKeyboardFromFont(myFont: Integer);
VAR
   theFontScript: Integer;

BEGIN
                              {get script code from font ID.}
   theFontScript := FontToScript(myFont);
                              {compare with keyboard script, }
                              { change if necessary}
   IF (GetScriptManagerVariable(smKeyScript) <>
                                        theFontScript) THEN
      KeyScript(theFontScript);
END;
```

Setting the Font Script From the Keyboard Script

Each time the user types a character other than a control character, your application should check that the font script is still the same as the keyboard script. The user may have, for example, switched keyboard scripts since entering the last character. If the font script does not match the keyboard script, change the current font to correspond to the new keyboard script before displaying the character. Follow these guidelines:

- If possible, set the current font to the previous font that was used for that script (that is, the last font for that script preceding the current point in the document or text buffer).
- Otherwise, set the font to one of the preferred fonts for that script system. The preferred fonts are the preferred application font, the preferred system font, the preferred monospaced font, and the preferred small font. (The ID numbers of these fonts can be obtained through the `GetScriptVariable` function.)

Listing 6-5 is an example of setting the font (and therefore the font script) from the keyboard script. It calls `GetScriptManagerVariable` with the `smKeyScript` selector to determine the current keyboard script. It then calls `FontToScript` to determine whether the keyboard script differs from the font script. If it does, the routine calls `GetScriptVariable` with the `smScriptAppFond` selector to determine the application font for the script. Then it sets the current font based on that result.

Listing 6-5 Setting the font (script) from the keyboard script

```
PROCEDURE MySetFontFromKeyboard(VAR myFont: Integer);
VAR
   scriptNum:  LongInt;

BEGIN
   scriptNum := GetScriptManagerVariable(smKeyScript);
   IF (FontToScript(myFont) <> scriptNum) THEN
      myFont := GetScriptVariable(scriptNum, smScriptAppFond);
   TextFont(myFont)
END;
```

You can also use this code if your application does not have an interface that lets users change fonts but still needs to provide for different script systems.

Obtaining Information

The second principal use for the Script Manager is in obtaining script-specific information. Many of the routines described in this section are of general interest and are used by most text applications. You can use these Script Manager routines to

- determine script codes for the current script system or any other available script system, based on font information
- analyze characters in your text for size (in bytes) or other properties
- directly access the contents of a script system's international resources, to pass that information to other text-handling calls or to inspect or modify the information

Most text-processing applications need script-code information and character-type information, and may need to pass specific tables from international resources to some script-aware text routines. If you format currencies, you need access to the numeric-format resource. If you use special symbols or if you format numbers, you need access to the untoken table and perhaps the number parts table of the tokens resource. If your needs are more specialized, you can obtain the contents of other tables and other resources.

Determining Script Codes From Font Information

The script management system asssociates a script system with a sequence of text by examining the font of that text. Your application may also need the same information—to test for the presence of a particular script system, to load its resources, to pass its code as a parameter to a script-aware routine, or to execute script-specific conditional code. You may need to determine what script system is currently active for displaying text, what script system is being used to sort and format text, or what script system would be used if text of a particular font were to be displayed or formatted. The Script Manager provides three routines for that purpose: `FontScript`, `FontToScript`, and `IntlScript`.

The `FontScript` function tells you which script system the font of the current graphics port belongs to. The `FontToScript` function tells you which (available) script system a font of any ID number belongs to. The `IntlScript` function tells you which script system is used by the Text Utilities to determine the number, date, time, currency, and sorting formats.

The `FontToScript` function returns a script code for a specified font family ID, but the `FontScript` and `IntlScript` functions return the code for the **current script,** the presently active script system for text manipulation. Many script-aware routines in QuickDraw, Text Utilities, the Script Manager, and other parts of the Macintosh script management system need not take an explicit script code or international resource handle as a parameter; in that case they use the current script as the script system under which they are to function.

The current script for text display is normally the font script. The current script for date and time formatting and string sorting is by default the system script. However, the settings of two flags—the font force flag and the international resources selection flag—can affect which script system is considered current at any one moment. Furthermore, if the mapping from font to script results in a request for a script system that is not available, the result defaults to the system script.

The next subsection lists the steps taken by `FontScript`, `FontToScript`, and `IntlScript` to determine the script codes they return, and the following subsections discuss the font force flag and the international resources selection flag in more detail.

How a Script Code Is Determined

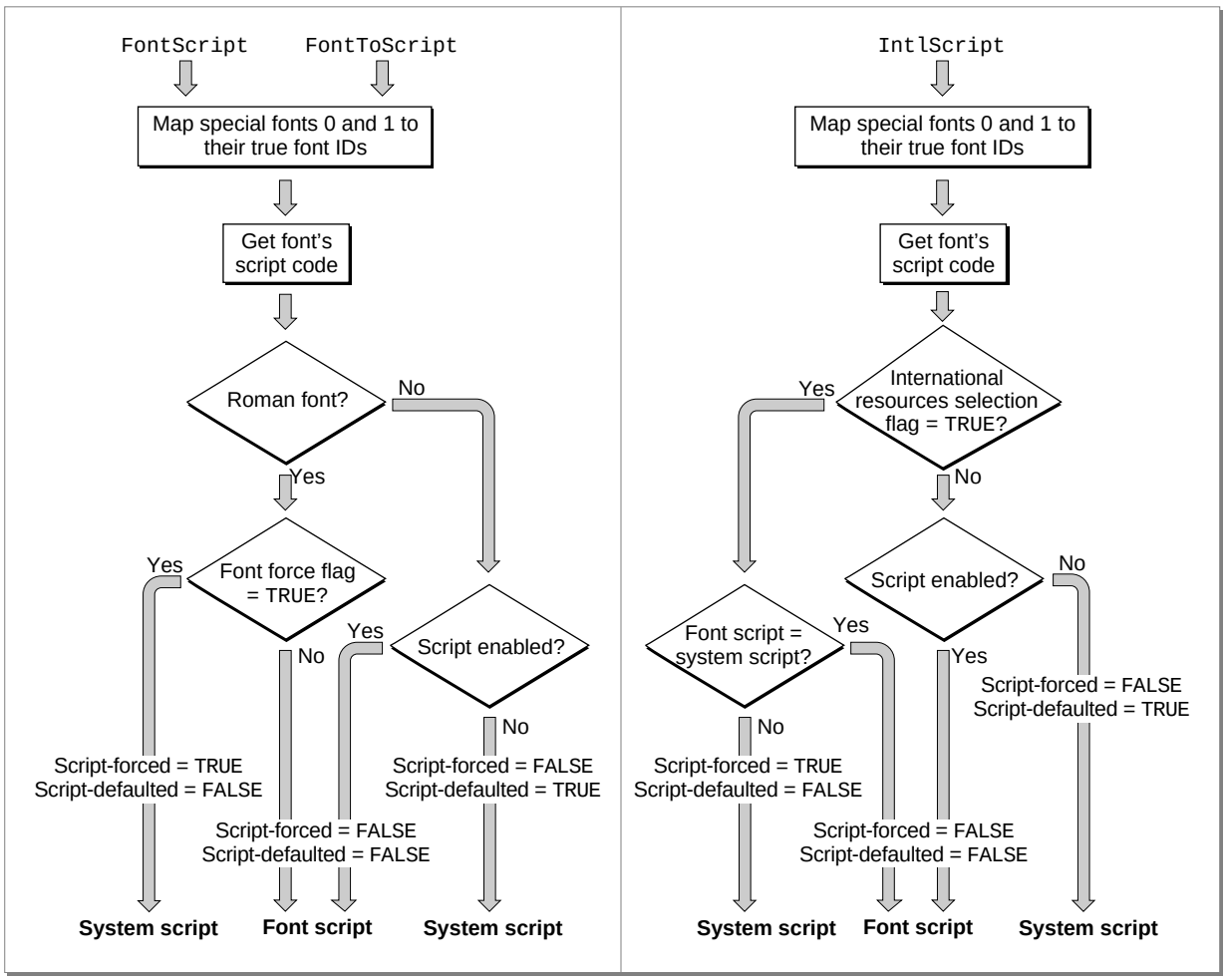
The `FontScript`, `FontToScript`, and `IntlScript` functions all use a font family ID to determine the script code they return. The formula they use is presented in the discussion of resource ID numbers and script codes in the appendix "International Resources" in this book. Fonts with IDs below 16384 ($4000) are all Roman; starting with 16384 each non-Roman script system has a range of 512 ($200) font IDs available.

Nevertheless, you should always call the functions instead of hardcoding any formula, because it may change in the future. Furthermore, the function results are influenced by the states of the font force flag and the international resources selection flag, and by the availability of the determined script. Figure 6-1 shows the method the functions follow:

1. The three functions initialize two result flags, the **script-forced result flag** and the **script-defaulted result flag,** to `FALSE`. These flags are Script Manager variables, accessed through the `GetScriptManagerVariable` function selectors `smForced` and `smDefault`.
2. The three functions map the two special font designations 0 and 1, meaning the system and application fonts, to their true font family ID numbers.
3. `FontScript` and `IntlScript` calculate the script code from the font family ID of the current font of the active port; `FontToScript` calculates the script code from the supplied font family ID. If the ID is in the range $4000 to $BFFF, it is a non-Roman font; otherwise, it is Roman.

4. Once the initial determination of the script code has been made, the three functions diverge:
   - If the font is Roman, `FontScript` and `FontToScript` examine the font force flag, which can be accessed through the `GetScriptManagerVariable` function selector `smFontForce`. If the flag is `TRUE`, the two functions substitute the system script for the font script, and set the script-forced result flag to `TRUE`. If the font is non-Roman, `FontScript` and `FontToScript` ignore the state of the font force flag.
   - Regardless of the font type (Roman or non-Roman), `IntlScript` examines the international resources selection flag, which can be accessed through the `GetScriptManagerVariable` function selector `smIntlForce`. If the flag is `TRUE` and the font script does not equal the system script, `IntlScript` substitutes the system script for the font script and sets the script-forced result flag to `TRUE`.

Figure 6-1 Determining script code from font family ID



5. A final check is made to be sure that the resulting script is installed and enabled. If it is not, the three functions substitute the system script for the script code previously determined, set the script-forced result flag to FALSE, and set the script-defaulted result flag to TRUE.
6. The functions return the resulting script code in their function results.

Call FontScript when you want to know which script system will be used for text layout and display. The script code returned by FontScript tells you which script system controls the functioning of such calls as CharToPixel, CharacterType, FindWordBreaks, DrawText, and DrawJustified. Typically, FontScript returns the script code for the font script; in most situations the font force flag is FALSE, because applications usually expect to format and draw text according to the rules of the font script.

Call FontToScript when you want to know whether the script system for text of a particular font is available, or when you wish to manipulate text of a certain script system without setting the current font to that font's ID.

Note
Because a user can set the value of the font force flag from the Text control panel, the result returned from the FontToScript or FontScript function for a font whose ID number is in the Roman range can vary from call to call. ◆

Call IntlScript when you want to know which script system will be used for formatting dates and numbers, and for sorting strings. The script code returned by IntlScript tells you which script system controls the functioning of such calls as DateString, LongTimeString, and CompareText, when no explicit script code or resource handle is supplied to those calls. In many localized versions of sysem software, IntlScript by default returns the script code for the system script, because the international resources selection flag is by default TRUE. The Finder and other parts of system software usually expect to present dates, times, and lists of files according to the rules of the system script.

Because the two flags are independent of each other, two different meanings for current script can exist simultaneously. For example, your application might be sorting a set of strings by one script's rules, but displaying them by another's. If that is not appropriate, set the flags as needed before formatting or drawing. See the following discussion.

Using the Font Force Flag

You access and control the font force flag through the GetScriptManagerVariable and SetScriptManagerVariable functions, with the selector smfontForce. This flag directly affects the results of the FontScript and FontToScript functions, and indirectly affects the operation of script-aware text measuring and drawing routines.

At startup, the Script Manager sets the font force flag to the value specified in the system script's international configuration ('itlc') resource. Typically, that value is FALSE.

When the font force flag is set to TRUE and the system script is non-Roman, the script management system interprets font family ID numbers in the range of the Roman script system ($0002 to $3FFF) as belonging to the system script instead. Character codes representing non-Roman characters in the system script are drawn using the system font instead of in the specified Roman font. This feature exists to allow users to enter and read non-Roman text in those few applications that have hardcoded font numbers.

For example, an application may hardcode Geneva as its font; it may force the `txFont` field of its graphics ports to always have a value of 3. (Note that this is a violation of good programming practice.) If the application is running on a system with Hebrew as the system script, it would normally be impossible to write properly in Hebrew because the hardcoded font ID would require the font script to be Roman. However, if the font force flag is set to TRUE, the script management system notes that the current font has an ID number in the Roman range and draws glyphs from the Hebrew system font for any character codes that represent valid Hebrew characters.

Thus to enter or read non-Roman text in these applications, the user can set the font force flag to TRUE from the Text control panel. Setting the font force flag is only partially effective, because it cannot give users full control over fonts. The user cannot choose, for example, which font belonging to the system script is to be substituted for Roman.

The font force flag has no effect on non-Roman fonts and has no effect if the system script is Roman. It affects only Roman fonts when the system script is non-Roman.

You can determine the status of font forcing by inspecting the script-forced result flag and the script-defaulted result flag immediately after calling `FontScript` or `FontToScript`; see Figure 6-1.

Although the font force flag exists primarily to accommodate restrictions in certain existing applications, it is a user-changeable setting that your application should be aware of and accommodate. For example:

- If you are writing any application in which the user has control over fonts, you should always set the font force flag to FALSE. There is no need to force fonts if the user can choose them.
- If the user sets the font force flag to TRUE, you will get the system script when you call `FontScript` or `FontToScript` for fonts in the Roman range, even if your application allows mixed text. To preserve Roman text, you can change the setting of the font force flag before calling `FontScript` or `FontToScript`, or before calling any other script-aware text routine. If you do that, be sure to save the previous value and restore it when your application exits or becomes inactive.

Using the International Resources Selection Flag

You access and control the international resources selection flag through the `GetScriptManagerVariable` and `SetScriptManagerVariable` functions, with the selector `smIntlForce`. This flag directly affects the results of the `IntlScript` function, and indirectly affects the operation of the `GetIntlResource` function and the script-aware Text Utilities sorting and formatting routines.

At startup, the Script Manager sets the international resources selection flag to the value specified in the system script's international configuration (`'itlc'`) resource. Typically, that value is `TRUE`.

The international resources selection flag affects the results of the `GetIntlResource` function (see page 6-90). `GetIntlResource` returns a handle to certain international resources, and the state of the international resources selection flag controls whether it is the system script or the font script whose international resources are loaded. When the flag is set to `TRUE`, `GetIntlResource` fetches the resources for the system script. When the flag is set to `FALSE`, `GetIntlResource` uses the current font in the active port to determine the script system whose resources will be fetched.

You can use the international resources selection flag to make sure that date formats, sorting, and so forth reflect the appropriate script in your application. Whenever you change the setting of the international resources selection flag, be sure to save the previous value and restore it when your application exits or becomes inactive.

Analyzing Characters

The Script Manager provides routines 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 script systems, you may need to know whether a particular character is a letter or a punctuation mark, whether or not it is uppercase, or whether it is part of a subscript (Roman within Cyrillic, Hiragana within Japanese, and so on).

Searching Text With Mixed Character Sizes

When searching for a single 1-byte character in text that may contain 2-byte characters, your application must not mistake part of a 2-byte character for the character you are seeking. The `CharacterByteType` and `FillParseTable` functions tell you whether a given character is 1-byte or whether it is the first or second byte of a 2-byte character.

These functions use the fact that, in a 2-byte script system, only a restricted set of values within the high-ASCII range are used as the first bytes of 2-byte characters, and those values are never used for 1-byte characters in that script system. All other byte values represent single-byte characters, control characters, or the second bytes of 2-byte characters. The ranges reserved for initial bytes of 2-byte characters vary from script system to script system, but every font has a table that gives that information, and `CharacterByteType` and `FillParseTable` use those tables to perform their calculations. For an illustration of this concept, see the discussion of character encoding in the chapter "Introduction to Text on the Macintosh" in this book.

Listing 6-6 shows a search procedure that accounts for 2-byte characters. This routine uses the Text Utilities `Munger` function to find a match to a key string. Because `Munger` might find a match beginning at the second byte of a 2-byte character, the routine checks for this case (using the `CharacterByteType` function) and continues searching if it occurs.

The sample assumes two application global variables: `gMainTextHandle`, which is a handle to the application's text buffer, and `gNewLocation`, a long-integer offset into the buffer at which to start searching. The parameters `keyPtr` and `keySize` specify the string to be matched in the text buffer; `scriptNum` is an explicit script code. On return, the routine updates `gNewLocation` to point to the location at which the search string was found, or sets it to –1 if no match was found.

Listing 6-6 Handling 2-byte characters in a search procedure

```
PROCEDURE MySearch (keyPtr: Ptr; keySize: LongInt; scriptNum:
Integer);
VAR
   byteType: Integer;
BEGIN
   HLock(gMainTextHandle);    {CharacterByteType can move memory}
   REPEAT BEGIN
      gNewLocation := Munger(gMainTextHandle, gNewLocation,
                             keyPtr, keySize, NIL, 0);
                             {if we matched second byte of }
                             { 2-byte char in text, continue}
      IF (gNewLocation >= 0) AND (scriptNum > 0) THEN
         byteType := CharacterByteType(gMainTextHandle^,
                                       gNewLocation, scriptNum)
      ELSE
         byteType := smSingleByte;
   END UNTIL byteType <> smLastByte;
   HUnlock(gMainTextHandle);
   IF (gNewLocation >= 0) AND {range-check, update global}
      (gNewLocation + keySize > GetHandleSize(gMainTextHandle))
THEN
      gNewLocation := -1;
END;
```

The `FillParseTable` function is similar to `CharacterByteType`, in that it helps you find 2-byte characters. However, you don't send `FillParseTable` the character code to be analyzed. Instead, `FillParseTable` fills in an entire 256-byte table of information for you, showing every byte value that is the first byte of a 2-byte character for the current font. You can use the table filled out by `FillParseTable` to find 2-byte characters in a large body of text much more rapidly than you could by calling `CharacterByteType` for each byte value in the text.

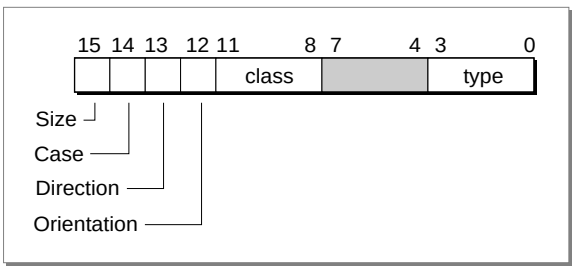
Getting Character-Type Information

You may want to know more about a byte than whether it is part of a 2-byte character. If you are simply searching for sequences of Roman text in a buffer, or if you wish to divide a run of Japanese into Kanji, Katakana, Hiragana, and Romaji components, you can use the `FindScriptRun` function described in the chapter "Text Utilities" in this book. But if you have other reasons to isolate specific types of characters, you can use `CharacterType`.

The `CharacterType` function is similar to `CharacterByteType`, in that it tells you what kind of character occurs at a given offset in a text buffer. But the kind of information it returns is different. `CharacterType` tells you what the character's line direction is, whether it's uppercase, whether it belongs to a subscript within its script, whether it's a 2-byte character, and what the character's specific type and class are—letter or punctuation, low-ASCII or high-ASCII Roman letter, Katakana or Hiragana, Jamo or Hangul, and so on.

When you call the `CharacterType` function, you pass it a byte offset; it returns a value that is an integer bit field giving information about the character at that offset. See Figure 6-2. The paragraphs following the figure describe the fields.

Figure 6-2 Fields in the `CharacterType` return value



Bits 0–3 of the `CharacterType` function result describe the **character type** of the character in question.

- The Roman script system recognizes three basic character types, defined by the following constants:

Character type	Hex. value	Explanation
`smCharPunct`	$0000	Punctuation (anything but a letter)
`smCharAscii`	$0001	ASCII letter (not a number or symbol, character code <= $7F)
`smCharExtAscii`	$0007	High-ASCII Roman letter (not a number or symbol, character code >= $80)

- Additional character-type constants are provided for Japanese Katakana and Hiragana; the ideographic subscripts such as Hanzi, Kanji, and Hanja; 2-byte Cyrillic and Greek in 2-byte systems; bidirectional script systems such as Arabic and Hebrew; and Korean Hangul and Jamo subscripts:

Character type	Hex. value	Explanation
`smCharKatakana`	$0002	Japanese Katakana
`smCharHiragana`	$0003	Japanese Hiragana
`smCharIdeographic`	$0004	Hanzi, Kanji, Hanja
`smCharTwoByteGreek`	$0005	2-byte Greek in 2-byte scripts
`smCharTwoByteRussian`	$0006	2-byte Cyrillic in 2-byte scripts
`smCharBidirect`	$0008	Arabic, Hebrew
`smCharContextualLR`	$0009	Thai, Indic, etc.
`smCharNonContextualLR`	$000A	Cyrillic, Greek, etc.
`smCharHangul`	$000C	Korean Hangul
`smCharJamo`	$000D	Korean Jamo
`smCharBopomofo`	$000E	Chinese Bopomofo (Zhuyinfuhao)

Bits 8–11 of the `CharacterType` function result describe the **character class** of the character in question. Character classes can be considered as subtypes of character types; a given character type can have several classes that belong to it.

- If the character type is `smCharPunct`, the following character classes are defined that include punctuation for both 1-byte and 2-byte script systems:

Character class	Hex. value	Explanation
`smPunctNormal`	$0000	Normal punctuation (such as ! , . ;?)
`smPunctNumber`	$0100	Number character (such as 0–9)
`smPunctSymbol`	$0200	Nonpunctuation symbol (such as # $ &)
`smPunctBlank`	$0300	Blank character (such as ASCII $00, $0D, $20)
`smPunctRepeat`	$0400	Repeat marker in 2-byte script
`smPunctGraphic`	$0500	Line graphics in 2-byte script

- In the Korean script system, if the character type is `smCharJamo`, the following character classes are defined. They determine whether a given byte contains a simple or complex consonant or a simple or complex vowel:

Character class	Hex. value	Explanation
`smJamoJaeum`	$0000	Simple consonant character
`smJamoBogJaeum`	$0100	Complex consonant character
`smJamoMoeum`	$0200	Simple vowel character
`smJamoBogMoeum`	$0300	Complex vowel character

The Jamo and Hangul subscripts of Korean are discussed briefly along with input methods in the chapter "Introduction to Text on the Macintosh" in this book.

- In the Japanese script system, if the character type is `smCharKatakana` or `smCharHiragana`, the following character classes are defined:

Character class	Hex. value	Explanation
	$0000	(none of the following defined classes)
`smKanaSmall`	$0001	Small Kana character
`smKanaHardOK`	$0002	Can have dakuten
`smKanaSoftOK`	$0003	Can have dakuten or han-dakuten

A small Kana character is a special form of Kana used to modify the pronunciation of a previous (full-sized) Kana character. Dakuten and han-dakuten are pronunciation marks that soften consonant sounds in Kana.

- In 2-byte script systems, if the character type is `smCharIdeographic`, the following character classes are defined:

Character class	Hex. value	Explanation
`smIdeographicLevel1`	$0000	Level 1 characters
`smIdeographicLevel2`	$0100	Level 2 characters
`smIdeographicUser`	$0200	User characters

The characters specified by the `smIdeographicLevel1` constant are part of the level 1 Han character set specified by Japanese, Chinese, and Korean government standards. Approximately 90 percent of normal text consists of characters from the level 1 set.

The characters specified by the `smIdeographicLevel2` constant are part of the level 2 Han character set, which includes obscure characters. The level 1 and level 2 character sets combined contain 98 percent of the character set used in the Kanji subscript.

The characters specified by `smIdeographicUser` represent custom characters created by the user.

Bits 12–15 of the `CharacterType` function result are the *character modifiers* of the character in question. One bit describes each modifier.

- Bit 12 specifies the *orientation* of the character: whether it is intended for horizontal or vertical writing.

Character orientation	Hex. value	Explanation
`smCharHorizontal`	$0000	Character form is for horizontal writing, or for both horizontal and vertical
`smCharVertical`	$1000	Character form is for vertical writing only

- Bit 13 specifies the *direction* of the character: whether its line direction is left-to-right or right-to-left.

Character direction	Hex. value	Explanation
`smCharLeft`	$0000	Character with left-to-right line direction
`smCharRight`	$2000	Character with right-to-left line direction

- Bit 14 specifies the *case* of the character: whether it is lowercase or uppercase.

Character case	Hex. value	Explanation
`smCharLower`	$0000	Lowercase character
`smCharUpper`	$4000	Uppercase character

- Bit 15 specifies the *size* of the character: whether it is 1 or 2 bytes long.

Character size	Hex. value	Explanation
`smChar1byte`	$0000	1-byte character
`smChar2byte`	$8000	2-byte character

You can describe individual characters with combinations of these constants. For example, if the byte being examined by `CharacterType` is a 1-byte English uppercase "A", then the value of the result could be expressed as `smChar1Byte` + `smCharUpper` + `smCharLeft` + `smCharASCII`. `CharacterType` indicates blank characters by a type `smCharPunct` and a class `smCharBlank`.

Some values are meaningful only in certain subscripts or script systems. The value `smCharUpper` is meaningless in a subscript that has no uppercase characters, for example; the value `smIdeographicLevel` is meaningless in 1-byte script systems.

You can use `CharacterType` for a variety of purposes—to validate input in numeric fields, to filter non-phonetic characters in an input method, or to search for punctuation, uppercase letters, and symbols. If you are breaking lines of text and are not using the Text Utilities `StyledLineBreak` function, you can use `CharacterType` to locate and skip whitespace characters at the ends of lines; see the description of text drawing in the chapter "QuickDraw Text" in this book.

The `CharacterType` function is described further on page 6-85.

Directly Accessing International Resources

This section shows how you can directly access the international resources of a script system. Such direct access can help you be more efficient in creating bilingual applications, formatting numbers in different scripts, accessing character information, and using tokens. Several script-aware Text Utilities calls can take a handle to an international resource as an input parameter; you can use the calls in this section to obtain those handles.

Your application can examine the international resources that determine numeric formats, date formats, string sorting, conversion to tokens, and character encoding or rendering by making the calls described here. You can also retrieve individual tables from some of the resources.

This access also helps you to provide your own versions or regional variations of certain international resources. See "Replacing a Script System's Default International Resources" beginning on page 6-48 for more information.

Note
Although you can access the international resources independently through the Resource Manager function `GetResource` and related calls, you can be sure to get the preferred resource of the current script system by using the calls described here. ◆

The calls you make to access the international resources are `ClearIntlResourceCache`, `GetIntlResource`, and `GetIntlResourceTable`. With them, you have access to the contents of a script system's numeric-format (`'itl0'`), long-date-format (`'itl1'`), string-manipulation (`'itl2'`), tokens (`'itl4'`), and encoding/rendering (`'itl5'`) resources.

To access one of these resources for the current script, follow these steps:

1. Make sure the current script is the script system containing the international resource you want to access. See "Determining Script Codes From Font Information" on page 6-21. You may need to verify the settings of the font script, the system script, and the international resources selection flag. See "Using the International Resources Selection Flag" on page 6-25.
2. If you need access to any version of the current script's string-manipulation or tokens resources other than its default version, call `ClearIntlResourceCache` first. See "Replacing a Script System's Default International Resources" on page 6-48.
3. Call `GetIntlResource`, specifying the type of resource you need. `GetIntlResource` returns a handle to the resource.

For an example of using `GetIntlResource` to extract information from an international resource, see the next section, "Using Currency, Number, and Date Formats."

To access a specific table within a string-manipulation or tokens resource, follow these steps:

1. If you don't already have it, determine the script code of the script system containing the international resource you want to access. See "Determining Script Codes From Font Information" on page 6-21.
2. If you need access to any other than the script's default version of that resource, call `ClearIntlResourceCache` first. See "Replacing a Script System's Default International Resources" on page 6-48.

3. Call `GetIntlResourceTable` to get the specified table within the specified resource belonging to the specified script system. Depending on the resource, you can get its number-parts, untoken, word-selection, line-break, or whitespace table.

For more information about these tables, see the following sections: "Using Number Parts," "Retrieving Text From Tokens," "Using Word-Break Tables," and "Using Whitespace Information."

IMPORTANT

Any time you replace the default international resources for a script system, whether or not you subsequently call `GetIntlResource` or `GetIntlResourceTable`, you need to call `ClearIntlResourceCache`, to make sure that the replacements are used by all script-aware calls. See "Replacing a Script System's Default International Resources" beginning on page 6-48. ▲

Using Currency, Number, and Date Formats

In general, you should use the Text Utilities routines for date, time, and number formatting. See the chapter "Text Utilities" in this book. If, however, you need to directly access fields in the numeric-format (`'itl0'`) and long-date-format (`'itl1'`) resources to find the characters, separators, strings, and orders for formatting numbers, dates, and times, you can do so with `GetIntlResource`.

Listing 6-7 shows how to determine the decimal, thousands, and list separators for number formatting in the current script. To access the numeric-format resource, the routine specifies a resource selector of 0 (for `'itl0'`) in the parameter `theID` of the `GetIntlResource` function. It then extracts the values it wants from the `decimalPt`, `thousSep`, and `listSep` fields.

Listing 6-7 Determining the number separators for the current script

```
PROCEDURE MyGetNumberSeparators (VAR myDecimal:Char;
                                 VAR myThousands:Char;
                                 VAR myListSep:Char);
VAR
   myHandle:      Intl0Hndl;
                          {make sure the desired script is set }
                          { before calling this routine}
BEGIN
   myHandle := Intl0Hndl(GetIntlResource(0));{Get 'itl0' resource}
   myDecimal := myHandle^^.decimalPt;  {for example, 1.234}
   myThousands := myHandle^^.thousSep; {for example, 1,234,567}
   myListSep := myHandle^^.listSep;    {for example, 1;2;3}
END;
```

IMPORTANT

Do not assume that the components of dates and times are always ordered in a left-to-right direction when displayed. If you are drawing individual time components, be careful not to simply draw them from left to right in all cases. For instance, the AM/PM characters in an English time string are on the right, whereas in an Arabic time string the equivalent characters may be on the left or right, depending on the primary line direction—even though in both cases these characters are at the end of the time string in memory. ▲

Using Number Parts

You can access information on how separators and other parts of formatted numbers are represented in a particular script system by examining the **number parts table** in the script's tokens (`'itl4'`) resource. Unlike the numeric-format resource, the number parts table supports 2-byte characters; it also contains more information, especially for complicated number formats such as scientific notation.

Your most common reason for obtaining the number parts table may be to pass it as a parameter to the Text Utilities functions `StringToFormatRec`, `FormatRecToString`, `StringToExtended`, and `ExtendedToString`. But you can also examine its contents. Listing 6-8 shows how to call the `GetIntlResourceTable` procedure, with a table selector of `smNumberPartsTable`, to obtain the number parts table associated with a given script. The routine obtains the character associated with the number part specified by `thePart` and saves it as a wide character, which is a character of either 1 or 2 bytes. (See the discussion of the tokens resource in the appendix "International Resources" for a definition of the `WideChar` data type.) To specify the system script, the parameter `theScript` would have the value `smSystemScript`. The parameter `thePart` can have such values as `tokDecPoint` and `tokThousands`. For a complete list of number-parts constants, see the description of the tokens resource in the appendix "International Resources" in this book.

Listing 6-8 Getting number parts from a script system's number parts table

```
PROCEDURE MyMapNumPartToWideChar(theScript: ScriptCode;
                                 thePart: Integer;
                                 VAR theWChar: WideChar);
VAR
   itlHandle: Handle;
   numpartsOffset: Longint;
   numpartsLength: LongInt;
   numpartsPtr: NumberPartsPtr;
```

```
BEGIN
   GetIntlResourceTable(theScript, smNumberPartsTable,
                        itlHandle, numpartsOffset,
                        numpartsLength);

   IF itlHandle = NIL THEN    {handle errors, }
      theWChar.b := 0         { return null WideChar}
   ELSE BEGIN                 {make numpartsPtr point to }
                              { beginning of number parts table}

      numpartsPtr := NumberPartsPtr(LongInt(itlHandle^) +
                     numpartsOffset);
      IF thePart > tokMaxSymbols THEN  {invalid number part-- }
                                       { handle  error, }
         theWChar.b := 0               { return null WideChar}
      ELSE BEGIN
         theWChar := numpartsPtr^.data[thePart];
      END;
   END;
END;
```

Retrieving Text From Tokens

Tokens are abstract entities that stand for classes of text items such as alphanumeric strings, various symbols, and quoted literals. The Script Manager `IntlTokenize` function converts programming-language text into script-independent tokens useful to compilers or interpreters. See "Tokenization" on page 6-38. The **untoken table** in a script system's tokens (`'itl4'`) resource has the opposite purpose; it helps you convert script-independent tokens into the text of a given script system.

The untoken table lists the characters associated with each fixed (invariant) token defined by that script. (An invariant token is one that, like `tokenColon`, represents a unique symbol. Other types of tokens, like `tokenAlpha`, represent an arbitrary sequence of characters.) If you need to find out, for example, how a given script system represents the "less than or equal to" symbol (is it the 1-byte character "≤", a 2-byte encoding of the character "≤", the 2-byte, 2-character sequence "<=", or something else altogether?), you can look up the values of `tokenLessEqual1` and `tokenLessEqual2` in that script's untoken table.

The untoken table is most useful for obtaining script-specific forms for individual common symbols, such as the ellipsis or center dot. If you truncate strings with the ellipsis character (…) or use the center dot (•) such as AppleShare does for echoing passwords, don't hardcode their character codes; they may not be valid in some script systems. Instead, specify `tokenEllipsis` or `tokenCenterDot`, and use the untoken table of the current script system to obtain the proper text for those tokens.

Note
If a script system has no defined character or string that corresponds to a particular token, the untoken table contains either a null string or the string "??" for that token. ◆

You access the untoken table by calling the `GetIntlResourceTable` procedure with a table selector of `smNumberPartsTable`. Listing 6-9 provides an example of how to access the untoken table in the tokens resource. This code sample extracts the canonical string associated with a token. It sets the parameter `theString` to the string that corresponds to the token `theToken`. (Usually, this string is 4 bytes or less.) To specify the system script, the parameter `theScript` would have the value `smSystemScript`. The parameter `theToken` can have such values as `tokenNoBreakSpace`, `tokenEllipsis`, and `tokenCenterDot`. For a complete list of defined constants for tokens, see "Token Codes" beginning on page 6-58.

Listing 6-9 Getting a token string from the untoken table

```
PROCEDURE MyMapTokenToString(theScript: ScriptCode; theToken:
Integer; VAR theString: Str255);

VAR
   itlHandle:        Handle;
   untokenOffset:    LongInt;
   untokenLength:    LongInt;
   untokenPtr:       UntokenTablePtr;
   untokenStringPtr: StringPtr;

BEGIN
   GetIntlResourceTable(theScript, smUnTokenTable, itlHandle,
                        untokenOffset, untokenLength);

   IF itlHandle = NIL THEN    {handle errors, return null string}
      theString := ''
   ELSE BEGIN                 {make untokenPtr point to the }
                              { beginning of the untoken table}
      untokenPtr := UntokenTablePtr(LongInt(itlHandle^) +
                    untokenOffset);
      IF theToken > untokenPtr^.lastToken THEN  {this token is }
                                          { not in table-- }
         theString := ''                  { return null string}
      ELSE BEGIN              {index[theToken] is the offset }
                              { of the desired string from the }
                              { beginning of the untoken table}
         untokenStringPtr := StringPtr(LongInt(untokenPtr) +
```

```
untokenPtr^.index[theToken]);
         theString := untokenStringPtr^;
      END;
   END;
END;
```

Even though using the untoken table is conceptually the converse of calling the `IntlTokenize` function, their purposes are different. `IntlTokenize` is used as a first step toward compiling or interpreting programming-language source text, and its results are rarely returned or reconverted to source text. The untoken table is most commonly used to supply localized text for individual common tokens.

Using Word-Break Tables

If you use the Text Utilities `FindWordBreaks` procedure to determine the boundaries of a word, you normally do not need to pass it an explicit pointer to a word-break table. However, if you want to use a custom word-break table you can pass `FindWordBreaks` a pointer to that table. Word-break tables are in a script system's string-manipulation (`'itl2'`) resource; you can gain access to them by calling the `GetIntlResourceTable` procedure with a table selector of `smWordSelectTable` or `smWordWrapTable`.

There are two possible table selectors because a script system may have different word breaks for word selection than it does for line breaking. If you are using `FindWordBreaks` to select an individual word, use `smWordSelectTable` when you call `GetIntlResourceTable` to obtain the word-break table. If you are using `FindWordBreaks` to find line breaks, use `smWordWrapTable` when you call `GetIntlResourceTable`.

Using Whitespace Information

Most applications that need whitespace information, such as when eliminating extra spaces in text or searching for non-space characters, can get it by calling the `CharacterType` function. However, if your application needs a listing of all valid whitespace characters in a script system, you can call `GetIntlResourceTable` with a table selector of `smWhiteSpaceList`. `GetIntlResourceTable` returns the whitespace table from the script system's tokens resource.

Converting Text

The third principal use for the Script Manager is in converting text from one form to another, for two specific purposes: tokenization and transliteration. The routines described in this section are used by specialized applications only. You can use these Script Manager routines to

- lexically convert text of the current script system into a series of language-independent tokens (tokenization)
- phonetically convert text of one subscript into text of another subscript within the same script system (transliteration)

Most text-processing applications have no need to perform either of these tasks. However, if your program needs to evaluate programming statements or logical or mathematical expressions in a script-independent fashion, you may want to use the Script Manager's tokenization facility. If your program performs phonetic conversion, for text input or for any other purpose, you may want to use the Script Manager's transliteration facility.

Tokenization

Programs that parse structured text expressions (such as compilers, assemblers, and scripting-language interpreters) usually assign sequences of characters to categories called **tokens.** Tokens are abstract entities that stand for names, operators, and quoted literals without making assumptions that depend on a particular writing system.

The Script Manager provides support for this conversion, called **tokenization.** Each script system's international tokens resource (type `'itl4'`) contains tables of token information used by the Script Manager's `IntlTokenize` function to identify the elements in an arbitrary string of text and convert them to tokens. The token stream created by `IntlTokenize` can be used as input to a compiler or interpreter, or to an expression evaluator such as might be used by a spreadsheet or database program.

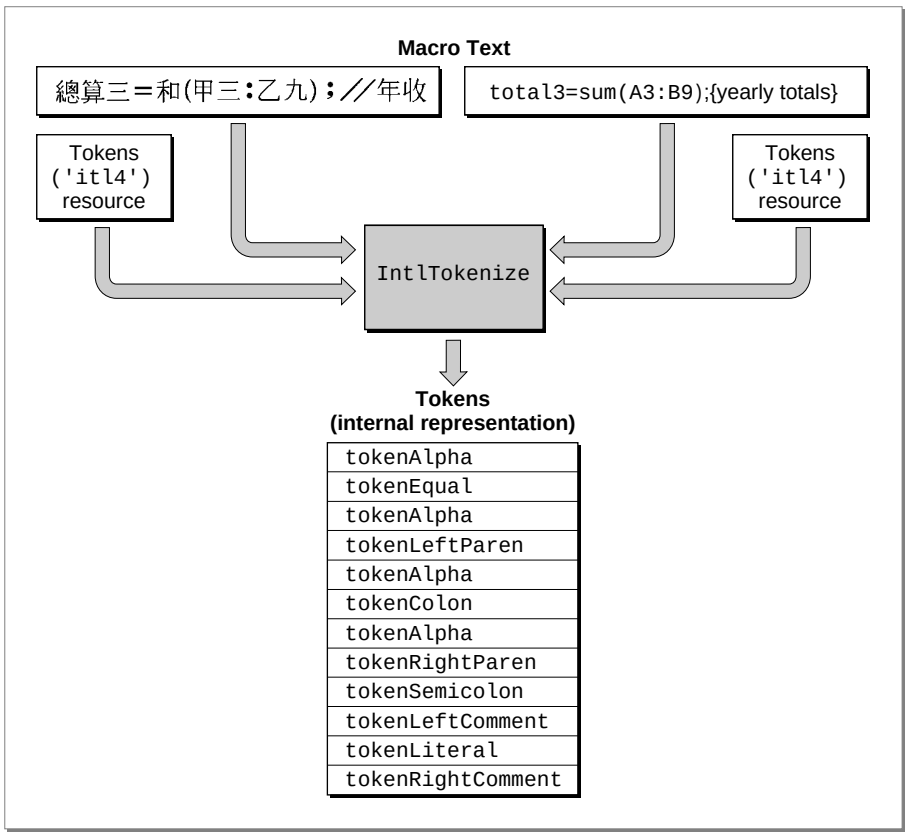
The `IntlTokenize` function allows your application to create a common set of tokens from text in any script system. For example, a whitespace character might have different character-code representations in different script systems. The `IntlTokenize` function can assign the token `tokenWhite` to any whitespace character, thus removing dependence on any character-encoding scheme.

When you call `IntlTokenize`, you pass it the source text to interpret. `IntlTokenize` parses the text and returns a list of the tokens that make up the text. Among the token types that it recognizes are whitespace characters; newline or return characters; sequences of alphabetic, numeric, and decimal characters; the end of a stream of characters; unknown characters; alternate digits and decimals; and many fixed token symbols, such as open parentheses, plus and minus signs, commas, and periods. See page 6-58 for a complete list of recognized tokens and their defined constants.

`IntlTokenize` can return not only a list of the token types found in your text but also a normalized copy of the text of each of the tokens, so that the content of your source text is preserved along with the tokens generated from it.

Figure 6-3 illustrates the process that occurs when `IntlTokenize` converts text into a sequence of tokens. It shows that very different text from two separate script systems can result in the same set of tokens.

Figure 6-3 The action of `IntlTokenize`



Because it uses the tokens resource belonging to the script system of the text being analyzed, `IntlTokenize` works on only one script run at a time. However, one way to process multiscript text is to make successive calls to `IntlTokenize` and append the results of each to the token list, thus building a single token stream from multiple calls.

Note

The `IntlTokenize` function does not provide complete lexical analysis; it returns a simple, sequential list of tokens. If necessary, your application can then process the output of `IntlTokenize` at a more sophisticated lexical or syntactic level. ◆

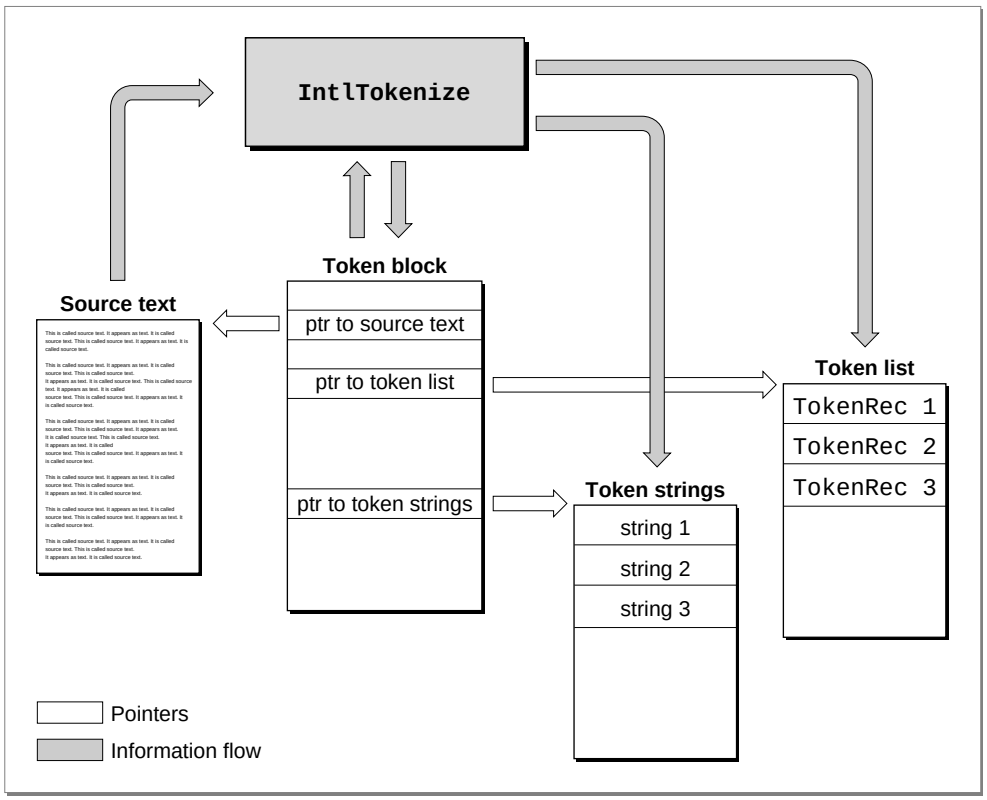
The rest of this section introduces the data structures used by `IntlTokenize`, discusses specific features and how it handles specific types of text, and gives an example.

Data Structures

When you call `IntlTokenize`, you supply it with a pointer to a **token block record,** a data structure that you have allocated. The token block record has a pointer to your source text and pointers to two other buffers you have allocated—one to hold the list of **token records** that `IntlTokenize` generates and the other to hold the string representations of those tokens, if you choose to have strings generated. See Figure 6-4.

`IntlTokenize` fills in the token list and the string list, updates information in the token block record, and returns the information to you.

Figure 6-4 `IntlTokenize` data structures (simplified)



Delimiters for Literals and Comments

Your application may specify up to two pairs of delimiters each for quoted literals and for comments. Quoted literal delimiters consist of a single symbol, and comment delimiters may be either one or two symbols (including the newline character for notations whose comments automatically terminate at the end of a line). Each delimiter is represented by a token, as is the entire literal between the opening and closing delimiters—except when the literal contains an escape character; see "Escape Character" (next).

Limited support exists for nested comments. Comments may be nested if so specified by the `doNest` flag, with one restriction that must be strictly observed to prevent `IntlTokenize` from malfunctioning: nesting is legal only if both the left and right delimiters for the comment token are composed of two symbols each. If your application specifies two different sets of comment delimiters, then the `doNest` flag always applies to both.

IMPORTANT
When using nested comments specified by the `doNest` flag, test thoroughly to ensure that the requirements of `IntlTokenize` are met. ▲

Escape Character

The characters that compose literals within quotations and comments are normally defined to have no syntactic significance; however, the escape character within a quoted literal signals that the following character should not be treated as the closing delimiter. Outside of the limits of a quoted literal, the escape character has no significance and is not recognized as an escape character.

For example, if the backslash "\" (token type = `tokenBackSlash`) is defined as the escape character, the `IntlTokenize` function would consider it to be an escape character in the following string, and would not consider the second quotation mark to be a closing delimiter:

```
"This is a quote \" within a quoted literal"
```

In the following string, however, `IntlTokenize` would not consider the backslash to be an escape character, and therefore would consider the first quotation mark to be an opening delimiter:

```
This is a backslash \" preceding a quoted literal"
```

Alphanumeric Tokens

The `IntlTokenize` function allows you to specify that numeric characters do not have to be considered numbers when mixed with alphabetic characters. If a flag is set, alphabetic sequences may include digits, as long as first character is alphabetic. In that case the sequence `Highway61` would be converted to a single alphabetic token, instead of the alphabetic token `Highway` followed by the number `61`.

Alternate Numerals

Some script systems have not only Western digits (that is, the standard ASCII digits, the numerals 0 through 9), but also their own numeral codes. `IntlTokenize` recognizes these alternate numerals and constructs tokens from them, such as `tokenAltNum` and `tokenAltReal`.

String Generation

To preserve the content of your source text as well as the tokens generated from it, your application may instruct `IntlTokenize` to generate null-terminated, even-byte-boundaried Pascal strings corresponding to each token. `IntlTokenize` constructs the strings according to these rules:

- If the token is anything but alphabetic or numeric, `IntlTokenize` copies the text of the token verbatim into the Pascal string.
- If the token represents non-Roman alphanumeric characters, `IntlTokenize` copies the characters verbatim into the Pascal string.
- If the token represents Roman alphabetic characters, `IntlTokenize` normalizes them to standard ASCII characters (such as by changing 2-byte Roman to 1-byte Roman) and writes them into the Pascal string.
- If the token represents numeric characters—even if the script system uses an alternate set of digits—`IntlTokenize` normalizes them into standard ASCII numerical digits, with a period as the decimal separator, and creates a string from the result. This allows users of other script systems to transparently use their own numerals or Roman characters for numbers or keywords.

The tokens resource includes a string-copy routine that performs the necessary string normalization.

Appending Results

You can make a series of calls to `IntlTokenize` and append the results of each call to the results of previous calls. You can instruct `IntlTokenize` to use the output values for certain parameters from each call as input values to the next call. At the end of your sequence of calls you will have—in order—all the tokens and strings generated from the calls to `IntlTokenize`.

Appending results is the only way to use `IntlTokenize` to parse a body of text that has been written in two or more different script systems. Because `IntlTokenize` can operate only on a single script run at a time, you must first divide your text into script runs and pass each script's character stream separately to `IntlTokenize`.

Example

Here is an example of how the `IntlTokenize` function breaks text into segments that that can be processed in a way that is meaningful in a particular script system. The source text is identical to that shown in Figure 6-3 on page 6-39. Assume that you send this programming-language statement to `IntlTokenize`:

```
total3=sum(A3:B9);{yearly totals}
```

`IntlTokenize` might convert that into the following sequence of tokens and token strings:

Token	Token string
`tokenAlpha`	`'total3'`
`tokenEqual`	`'='`
`tokenAlpha`	`'sum'`
`tokenLeftParen`	`'('`
`tokenAlpha`	`'A3'`
`tokenColon`	`':'`
`tokenAlpha`	`'B9'`
`tokenRightParen`	`')'`
`tokenSemicolon`	`';'`
`tokenLeftComment`	`'{'`
`tokenLiteral`	`'yearly totals'`
`tokenRightComment`	`'}'`

This token sequence could then be processed meaningfully by an expression evaluator. If the statement had been created under a different script system, in which comment delimiters, semicolons, or equality were represented with different character codes, the resulting token sequence would still be the same and could be evaluated identically—although the strings generated from the tokens would be different.

The `IntlTokenize` function is described further on page 6-92.

Transliteration

The Script Manager provides support for **transliteration,** the automatic conversion of text from one form to another within a single script system. In the Roman script system, transliteration simply means case conversion. In Japanese, Chinese, and Korean script systems, it means the phonetic conversion of characters from one subscript to another.

The `TransliterateText` function performs the conversions. Tables that control transliteration for a 1-byte script system are in its international string-manipulation (`'itl2'`) resource; the tables for a 2-byte script system are in the script's transliteration (`'trsl'`) resource. This illustrates the difference in the meaning of transliteration for the two types of script systems: case conversion information is in the string-manipulation resource, whereas information needed for phonetic conversion is in the transliteration resource. The transliteration resource is available to all script systems, although currently no 1-byte script systems make use of it.

Transliteration here does not mean translation; the Macintosh script management system cannot translate text from one language to another. Nor does it include context-sensitive conversion from one subscript to another; that can be accomplished with an input method. See, for example, the discussions of input methods in the chapters "Introduction to Text on the Macintosh" and "Text Services Manager" in this book. Transliteration can, however, be an initial step for those more complex conversions:

- Within the Japanese script system, you can transliterate from Hiragana to Romaji (Roman) and from Romaji to Katakana, and vice versa. You cannot transliterate from Hiragana to Kanji (Chinese characters). However, transliteration from Romaji to Katakana or Hiragana could be an initial step for an input method that would complete the context-sensitive conversion to Kanji.
- Within the (traditional) Chinese script system, you can transliterate from the Bopomofo or Zhuyinfuhao (phonetic) subscript to Roman, and vice versa. You cannot transliterate from Zhuyinfuhao to Hanzi (Chinese characters). However, transliteration from Zhuyinfuhao to Pinyin could be an initial step for an input method that would complete the context-sensitive conversion to Hanzi.
- Within the Korean script system, you can transliterate from Roman to Jamo, from Jamo to Hangul, from Hangul to Jamo, and from Jamo to Roman. It is therefore possible to transliterate from Hangul to Roman and from Roman to Hangul by a two-step process. It is not possible to transliterate from Hangul into Hanja (Chinese characters). Transliteration from Jamo to Hangul is used by the input method supplied with the Korean script system; that transliteration is sufficient when Hanja characters are not used. To include Hanja characters requires additional context-sensitive processing by the input method.

The Script Manager defines two basic types of transliteration you can perform: conversion to Roman characters, and conversion to a native subscript within the same non-Roman script system. Within those categories there are subtypes. For instance, in Roman text, case conversion can be either to uppercase or to lowercase; in Japanese text, native conversion can be to Romaji, Hiragana, or Katakana.

You can specify which types of text can undergo conversion. For example, in Japanese text you can, if you want to, limit transliteration to Hiragana characters only. Or you can restrict it to case conversion of Roman characters only.

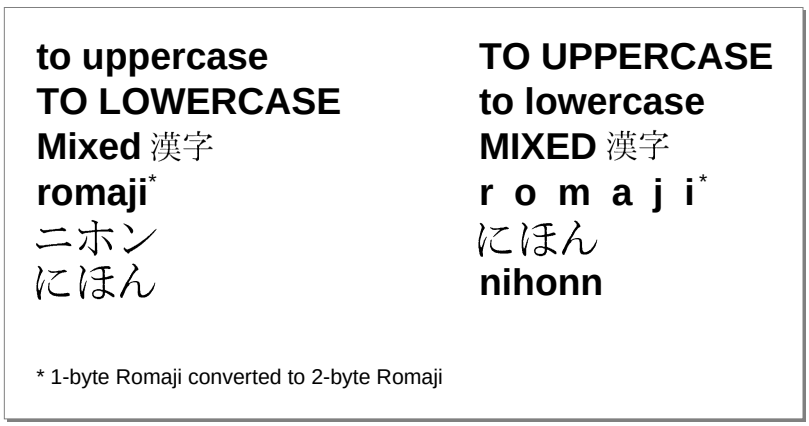
Not all combinations of transliteration are possible, of course. Case conversion cannot take place in scripts or subscripts that do not have case; transliteration from one subscript to another cannot take place in scripts that do not have subscripts.

Transliteration is not perfect. Typically, it gives a unique result within a 2-byte script, although it may not always be the most phonetic or natural result. Transliterations may be incorrect in ambiguous situations; by analogy, in certain transliterations from English "th" could refer to the sound in *the,* the sound in *thick,* or the sounds in *boathouse*.

Figure 6-5 shows some of the possible effects of transliteration. Each string on the right side of the figure is the transliterated result of its equivalent string on the left.

- Roman characters can be transposed from uppercase to lowercase and vice versa—even if they are embedded in text that also contains Kanji.
- One-byte Roman characters can be converted to 2-byte Roman characters. (The glyphs for 2-byte Roman characters are typically larger and spaced farther apart, for better appearance when interspersed with ideographic glyphs.)
- Katakana can be converted to Hiragana.
- Hiragana can be converted to 1-byte Roman characters.

Figure 6-5 The effects of transliteration



When you call `TransliterateText`, you specify a **source mask,** a **target format,** and a **target modifier.** The source mask specifies which subscript or subscripts represented in the source text should be converted to the target format. The target modifier provides additional formatting instructions. For example, in Japanese text that contains Roman, Hiragana, Katakana, and Kanji characters, you could use the source mask to limit transliteration to Hiragana characters only. You could then use the target format to specify conversion to Roman, and you could use the target modifier to further specify that the converted text become uppercase.

For all script systems, there are three currently defined values for source mask, with the following assigned constants:

Source mask constant	Value	Explanation
`smMaskAscii`	1	Convert from Roman text
`smMaskNative`	2	Convert from text native to current script
`smMaskAll`	–1	Convert from all text

To specify that you want to convert only Roman characters, use `smMaskAscii`. To convert only native characters, use `smMaskNative`. Use the `smMaskAll` constant to specify that you want to transliterate all text. "Roman text" is defined as any Roman characters in the character set of a given script system. In most cases, this means the low-ASCII Roman characters, but—depending on the script system—it may also include certain characters in the high-ASCII range whose codes are not used for the script system's native character set, and it may include 2-byte Roman characters in 2-byte script systems. The definition of "native text" is also script-dependent.

The 2-byte script systems recognize the following additional values for source mask:

Source mask constant	Hex. value	Explanation
All 2-byte scripts:		
`smMaskAscii1`	$04	Convert from 1-byte Roman text
`smMaskAscii2`	$08	Convert from 2-byte Roman text
Japanese:		
`smMaskKana1`	$10	Convert from 1-byte Katakana text
`smMaskKana2`	$20	Convert from 2-byte Katakana text
`smMaskGana2`	$80	Convert from 2-byte Hiragana text
Korean:		
`smMaskHangul2`	$100	Convert from 2-byte Hangul text
`smMaskJamo2`	$200	Convert from 2-byte Jamo text
Chinese:		
`smMaskBopomofo2`	$400	Convert from 2-byte Zhuyinfuhao text

The low-order byte of the `target` parameter is the format; it determines what form the text should be transliterated to. For all script systems, there are two currently supported values for target format, with the following assigned constants:

Target format constant	Hex. value	Explanation
`smTransAscii`	$00	Convert to Roman
`smTransNative`	$01	Convert to a subscript native to current script
`smTransCase`	$FE	Convert case for all text (obsolete)
`smTransSystem`	$FF	Convert to system script (obsolete)

The 2-byte script systems recognize the following additional values for target format:

Target format constant	Value	Explanation
All 2-byte scripts:		
`smTransASCII1`	2	Convert to 1-byte Roman text
`smTransASCII2`	3	Convert to 2-byte Roman text
Japanese:		
`smTransKana1`	4	Convert to 1-byte Katakana text
`smTransKana2`	5	Convert to 2-byte Katakana text
`smTransGana2`	7	Convert to 2-byte Hiragana text
Korean:		
`smTransHangul2`	8	Convert to 2-byte Hangul text
`smTransJamo2`	9	Convert to 2-byte Jamo text
Chinese:		
`smTransBopomofo2`	10	Convert to 2-byte Zhuyinfuhao text

The high-order byte of the `target` parameter is the target modifier; it provides additional formatting instructions. All script systems recognize these values for target modifier, with the following assigned constants:

Target modifier constant	Hex. value	Explanation
`smTransLower`	$4000	Target becomes lowercase
`smTransUpper`	$8000	Target becomes uppercase

For example, for `TransliterateText` to convert all the characters in a block of text to 1-byte Roman uppercase, the value of `srcMask` is `smMaskAll` and the target value is `smTransAscii1`+`smTransUpper`. To convert only those characters that are already (1-byte or 2-byte) Roman, the value of `srcMask` is `smMaskAscii1+smMaskAscii2`.

The `TransliterateText` function is described further on page 6-98.

Note
For uppercasing or lowercasing Roman text in general, use `UppercaseText` or `LowercaseText`. Because the performance of `TransliterateText` is slower, you may rarely want to use its case-changing capabilities in Roman text. ◆

Modifying Script Systems

The fourth principal use for the Script Manager is in modifying the contents of script systems themselves. The routines described in this section are for specialized purposes, such as providing regional variants to existing script systems or assigning script-specific features to individual documents or applications. You can use these Script Manager routines to

- replace one or more of a script system's international resources (this replacement occurs within the context of your application only)
- replace one or more of an individual script system's routines (for 1-byte complex scripts only)

Most text-processing applications need not perform either of these replacements. However, if your program has special needs or if you are implementing a specific regional variation of a script system with unusual text-handling features, you can use these Script Manager calls.

Replacing a Script System's Default International Resources

In certain situations, you may want to replace the script-system-supplied international resources with some of your own. For example, your application might create documents containing currency amounts and get the currency format from the numeric-format resource. You may then want the unit of currency to remain the same, even if the document is displayed on a Macintosh with system software localized for another region.

You can store your own versions of some of the international resources in your application's or document's resource file, to override those in the System file. In this case, documents that your application creates could have their own copy of the numeric-format resource that was used to create them.

To replace the numeric-format (`'itl0'`) or long-date-format (`'itl1'`) resource, follow these two steps:

1. When your application starts up or when your document is opened, call the `GetScriptVariable` function for your target script system to get the ID number of the current default version of the resource you are replacing. Save that ID number for later.
2. Call the `SetScriptVariable` function to set the script's default ID number to the ID of the resource that you are supplying.

If your replacement resource is attached to your application or document, it will override the script system's default version. When a call for a resource is made, the Resource Manager searches first in the resource fork of the open document, then in the resource fork of the active application, and then in the System file. This search sequence is described in the chapter "Resource Manager" in *Inside Macintosh: More Macintosh Toolbox.*

To substitute the string-manipulation (`'itl2'`) or tokens (`'itl4'`) resources, you must take an additional step. If you want to replace the default resource currently used by a script system, you must first clear your application's **international resources cache.** The cache is part of an application's context as handled by the Process Manager; it is initialized when the application is launched, and is switched in and out with the application. It contains the resource ID numbers of the default string-manipulation and tokens resources for all installed script systems. Once the cache is set up, access to string-manipulation and tokens resources is exclusively through the ID numbers in the cache.

Therefore, to replace a string-manipulation or tokens resource, it is not enough to attach the resource to your document and call the `SetScriptVariable` function; this alone does not affect any cached ID numbers. In addition, add this third step:

3. After calling `SetScriptVariable` as described in step 2, call `ClearIntlResourceCache`. That will cause the cache to be reloaded with the current default resource ID numbers, including your override of the previous default, as each resource is called.

In this case, when a call for a resource is made, the Script Manager looks first in the cache for the resource ID to use. If the cache has been cleared, the Script Manager gets the ID from the script variables (and updates the cache with the new ID). The Script Manager then calls the Resource Manager, requesting a resource with that ID. The Resource Manager searches for the resource as described previously, taking it from your document or application.

Because the system maintains a separate international resources cache for each application's context, your application can provide its own string-manipulation and tokens resources without affecting the use of those resources by other applications or by the system. When the Process Manager switches in another application, that application's international resources cache has the defaults needed by that application.

No matter which international resource you have replaced, there is one final step to take:

4. When your application exits or is switched out, be sure to call `SetScriptVariable` once again to reset the script system's default ID number to what it was before you replaced it.

IMPORTANT

If the international resources selection flag is `TRUE` when a call to access your supplied resources is made, the ID numbers of your supplied resources must be within the system script range; if it is `FALSE`, the IDs must be in the range of the current script. Otherwise, your resources will not be found. See the appendix "International Resources" for a list of script codes and their resource ID ranges. ▲

Replacing a Script System's Default Routines

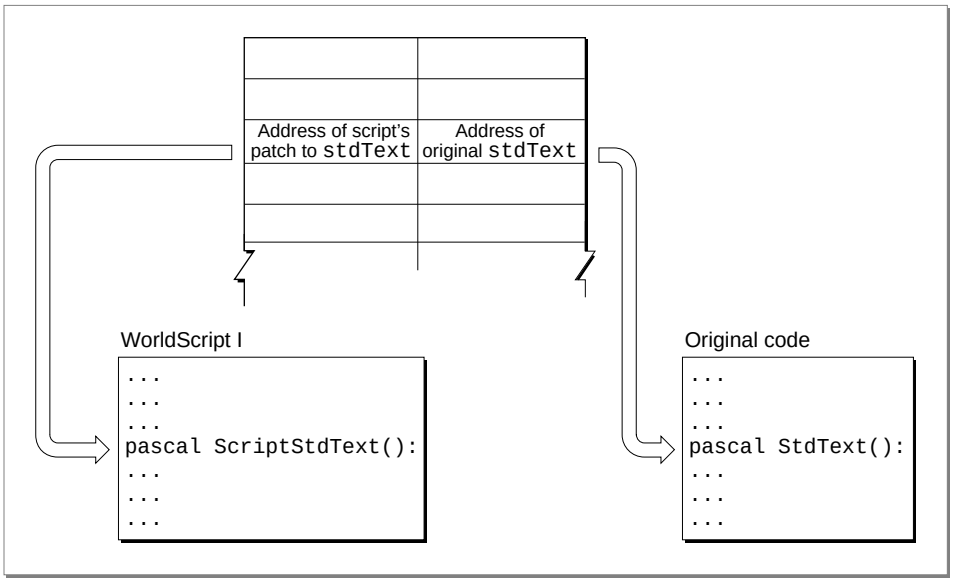
Applications do not normally need to modify a script system's text-handling routines. For 1-byte complex script systems and for 2-byte script systems, most script-specific behavior is built into tables in the script's international resources. Text-handling code is in Macintosh system software: in ROM, in the System file, or in one of two system extensions—WorldScript I and WorldScript II. WorldScript I and WorldScript II handle text for 1-byte complex and 2-byte script systems, respectively. They are described in the appendix "Built-in Script Support" in this book. For most needs the table-driven behavior is adequate, and you can access many of the tables through the Script Manager calls described in the previous section.

Even so, for 1-byte complex script systems, the Script Manager offers you the ability to modify or enhance the routines contained in WorldScript I. If you need specific script-based behavior that is not currently supported, you can replace one or more **script utilities** (low-level text-handling routines that employ the `_ScriptUtil` trap) or QuickDraw patches for your target script system. You can create a patch and install it with a System extension file (type `'INIT'`) that is executed at system startup.

IMPORTANT
Because this capability affects WorldScript I only, it is available only for 1-byte complex script systems. ▲

In every script system that uses WorldScript I, the dispatch-table element for every script utility and QuickDraw patch consists of two pointers: one to the WorldScript I implementation of the routine and one to the original (built-in Roman) routine. In all cases, the WorldScript I routine executes first. In some cases, WorldScript I calls the original routine after its own; in other cases, the pointer to the original routine is `NIL` and the WorldScript I routine is all that executes. See Figure 6-6. This design allows you to place a patched routine so that it executes before, in place of, or after the WorldScript I routine.

Figure 6-6 Dispatch table entry for script utilities and QuickDraw patches



The script-based dispatch table design gives you a simple, flexible way to replace individual routines without having to patch out all of `_ScriptUtil` or any of the QuickDraw low-level routines in their entirety. Furthermore, in a multiscript environment each patch applies only to its own script system. You can, for example, patch `stdText` for the Thai script system only, leaving it unchanged for all other script systems.

To replace only the WorldScript I implementation of a routine, replace its pointer in the dispatch table; to keep the WorldScript I routine while replacing or patching the original routine, replace the original-routine pointer in the dispatch table. The four Script Manager routines that allow you to make those patches are `GetScriptUtilityAddress`, `SetScriptUtilityAddress`, `GetScriptQDPatchAddress`, and `SetScriptQDPatchAddress`. Either pointer in the dispatch table may be `NIL`, meaning that WorldScript I either (1) doesn't patch the original routine or (2) doesn't call the original routine.

For additional information on how to use these four Script Manager routines to customize a script's behavior, see the appendix "Built-in Script Support."

Script Manager Reference

This section describes the constants, data structures, and routines that are specific to the Script Manager.

Constants

The Script Manager defines a large number of constants. This section lists and describes the constants with which you can specify

- script codes, language codes, and region codes
- token codes
- selectors for Script Manager variables
- selectors for script variables

There are many other constants defined for other purposes that are listed with the routines that use them. In addition, all constants are listed in the section "Summary of the Script Manager" beginning on page 6-107.

Script Codes

You can specify script systems with implicit and explicit script code constants in the `script` parameter of the `GetScriptVariable` and `SetScriptVariable` functions. The implicit script codes `smSystemScript` and `smCurrentScript` are special negative values for the system script and the font script, respectively.

Script constant	Value	Explanation
`smSystemScript`	–1	System script
`smCurrentScript`	–2	Font script
`smRoman`	0	Roman
`smJapanese`	1	Japanese
`smTradChinese`	2	Traditional Chinese
`smKorean`	3	Korean
`smArabic`	4	Arabic
`smHebrew`	5	Hebrew
`smGreek`	6	Greek
`smCyrillic`	7	Cyrillic
`smRSymbol`	8	Right-to-left symbols
`smDevanagari`	9	Devanagari

Script constant	Value	Explanation (continued)
smGurmukhi	10	Gurmukhi
smGujarati	11	Gujarati
smOriya	12	Oriya
smBengali	13	Bengali
smTamil	14	Tamil
smTelugu	15	Telugu
smKannada	16	Kannada/Kanarese
smMalayalam	17	Malayalam
smSinhalese	18	Sinhalese
smBurmese	19	Burmese
smKhmer	20	Khmer
smThai	21	Thai
smLaotian	22	Laotian
smGeorgian	23	Georgian
smArmenian	24	Armenian
smSimpChinese	25	Simplified Chinese
smTibetan	26	Tibetan
smMongolian	27	Mongolian
smGeez	28	Geez/Ethiopic
smEthiopic	28	= smGeez
smEastEurRoman	29	Extended Roman for Slavic and Baltic languages
smVietnamese	30	Extended Roman for Vietnamese
smExtArabic	31	Extended Arabic for Sindhi
smUninterp	32	Uninterpreted symbols

Note
The script code represented by the constant `smUninterp` is available for representation of special symbols, such as items in a tool palette, that must not be considered as part of any actual script system. For manipulating and drawing such symbols, the `smUninterp` constant should be treated as if it indicated the Roman script system rather than the system script; that is, the default behavior of uninterpreted symbols should be Roman. ◆

Note
The script code represented by the constant `smRSymbol` is available as an alternative to `smUninterp`, for representation of special symbols that have a right-to-left line direction. Note, however, that the script management system provides no direct support for representation of text with this script code. ◆

Language Codes

Language codes have the following defined values. Note that each language is associated with a script code.

Language constant	Value	Language	(Script code)
langEnglish	0	English	(smRoman)
langFrench	1	French	(smRoman)
langGerman	2	German	(smRoman)
langItalian	3	Italian	(smRoman)
langDutch	4	Dutch	(smRoman)
langSwedish	5	Swedish	(smRoman)
langSpanish	6	Spanish	(smRoman)
langDanish	7	Danish	(smRoman)
langPortuguese	8	Portuguese	(smRoman)
langNorwegian	9	Norwegian	(smRoman)
langHebrew	10	Hebrew	(smHebrew)
langJapanese	11	Japanese	(smJapanese)
langArabic	12	Arabic	(smArabic)
langFinnish	13	Finnish	(smRoman)
langGreek	14	Greek	(smGreek)
langIcelandic	15	Icelandic	(smRoman)
langMaltese	16	Maltese	(smRoman)
langTurkish	17	Turkish	(smRoman)
langCroatian	18	Croatian	(smRoman)
langTradChinese	19	Chinese (traditional chars.)	(smTradChinese)
langUrdu	20	Urdu	(smArabic)
langHindi	21	Hindi	(smDevanagari)
langThai	22	Thai	(smThai)
langKorean	23	Korean	(smKorean)
langLithuanian	24	Lithuanian	(smEastEurRoman)
langPolish	25	Polish	(smEastEurRoman)
langHungarian	26	Hungarian	(smEastEurRoman)
langEstonian	27	Estonian	(smEastEurRoman)
langLettish	28	Lettish	(smEastEurRoman)
langLatvian	28	= langLettish	
langSaamisk	29	(language of Lapps/Sami)	(smRoman)
langLappish	29	= langSaamisk	
langFaeroese	30	Faeroese	(smRoman)

Language constant	Value	Language	(Script code) (continued)
langFarsi	31	Farsi	(smArabic)
langPersian	31	= langFarsi	
langRussian	32	Russian	(smCyrillic)
langSimpChinese	33	Chinese (simplified chars.)	(smSimpChinese)
langFlemish	34	Flemish	(smRoman)
langIrish	35	Irish	(smRoman)
langAlbanian	36	Albanian	(smRoman)
langRomanian	37	Romanian	(smEastEurRoman)
langCzech	38	Czech	(smEastEurRoman)
langSlovak	39	Slovak	(smEastEurRoman)
langSlovenian	40	Slovenian	(smEastEurRoman)
langYiddish	41	Yiddish	(smHebrew)
langSerbian	42	Serbian	(smCyrillic)
langMacedonian	43	Macedonian	(smCyrillic)
langBulgarian	44	Bulgarian	(smCyrillic)
langUkrainian	45	Ukrainian	(smCyrillic)
langByelorussian	46	Byelorussian	(smCyrillic)
langUzbek	47	Uzbek	(smCyrillic)
langKazakh	48	Kazakh	(smCyrillic)
langAzerbaijani	49	Azerbaijani	(smCyrillic)
langAzerbaijanAr	50	Azerbaijani	(smArabic)
langArmenian	51	Armenian	(smArmenian)
langGeorgian	52	Georgian	(smGeorgian)
langMoldovan	53	Moldovan	(smCyrillic)
langMoldavian	53	= langMoldovan	(smCyrillic)
langKirghiz	54	Kirghiz	(smCyrillic)
langTajiki	55	Tajiki	(smCyrillic)
langTurkmen	56	Turkmen	(smCyrillic)
langMongolian	57	Mongolian	(smMongolian)
langMongolianCyr	58	Mongolian	(smCyrillic)
langPashto	59	Pashto	(smArabic)
langKurdish	60	Kurdish	(smArabic)
langKashmiri	61	Kashmiri	(smArabic)
langSindhi	62	Sindhi	(smExtArabic)
langTibetan	63	Tibetan	(smTibetan)

continued

Language constant	Value	Language	(Script code) (continued)
langNepali	64	Nepali	(smDevanagari)
langSanskrit	65	Sanskrit	(smDevanagari)
langMarathi	66	Marathi	(smDevanagari)
langBengali	67	Bengali	(smBengali)
langAssamese	68	Assamese	(smBengali)
langGujarati	69	Gujarati	(smGujarati)
langPunjabi	70	Punjabi	(smGurmukhi)
langOriya	71	Oriya	(smOriya)
langMalayalam	72	Malayalam	(smMalayalam)
langKannada	73	Kannada	(smKannada)
langTamil	74	Tamil	(smTamil)
langTelugu	75	Telugu	(smTelugu)
langSinhalese	76	Sinhalese	(smSinhalese)
langBurmese	77	Burmese	(smBurmese)
langKhmer	78	Khmer	(smKhmer)
langLao	79	Lao	(smLaotian)
langVietnamese	80	Vietnamese	(smVietnamese)
langIndonesian	81	Indonesian	(smRoman)
langTagalog	82	Tagalog	(smRoman)
langMalayRoman	83	Malay	(smRoman)
langMalayArabic	84	Malay	(smArabic)
langAmharic	85	Amharic	(smEthiopic)
langTigrinya	86	Tigrinya	(smEthiopic)
langGalla	87	Galla	(smEthiopic)
langOromo	87	= langGalla	
langSomali	88	Somali	(smRoman)
langSwahili	89	Swahili	(smRoman)
langRuanda	90	Ruanda	(smRoman)
langRundi	91	Rundi	(smRoman)
langChewa	92	Chewa	(smRoman)
langMalagasy	93	Malagasy	(smRoman)
langEsperanto	94	Esperanto	(mod. smRoman)
langWelsh	128	Welsh	(smRoman)
langBasque	129	Basque	(smRoman)
langCatalan	130	Catalan	(smRoman)
langLatin	131	Latin	(smRoman)

Language constant	Value	Language	(Script code) (continued)
langQuechua	132	Quechua	(smRoman)
langGuarani	133	Guarani	(smRoman)
langAymara	134	Aymara	(smRoman)
langTatar	135	Tatar	(smCyrillic)
langUighur	136	Uighur	(smArabic)
langDzongkha	137	Bhutanese	(smTibetan)
langJavaneseRom	138	Javanese	(smRoman)
langSundaneseRom	139	Sundanese	(smRoman)

Region Codes

Region codes have the following defined values. Each region is associated with a particular language code and script code (not shown). Note that the existence of a defined region code does not necessarily imply the existence of a version of Macintosh system software localized for that region.

Region constant	Value	Explanation
verUS	0	United States
verFrance	1	France
verBritain	2	Great Britain
verGermany	3	Germany
verItaly	4	Italy
verNetherlands	5	Netherlands
verFrBelgiumLux	6	French for Belgium and Luxembourg
verSweden	7	Sweden
verDenmark	9	Denmark
verPortugal	10	Portugal
verFrCanada	11	French Canada
verIsrael	13	Israel
verJapan	14	Japan
verAustralia	15	Australia
verArabia	16	the Arabic world
verArabic	16	= verArabia
verFinland	17	Finland
verFrSwiss	18	French for Switzerland
verGrSwiss	19	German for Switzerland

continued

Region constant	Value	Explanation (continued)
verGreece	20	Greece
verIceland	21	Iceland
verMalta	22	Malta
verCyprus	23	Cyprus
verTurkey	24	Turkey
verYugoCroatian	25	Croatian system for Yugoslavia
verIndiaHindi	33	Hindi system for India
verPakistan	34	Pakistan
verLithuania	41	Lithuania
verPoland	42	Poland
verHungary	43	Hungary
verEstonia	44	Estonia
verLatvia	45	Latvia
verLapland	46	Lapland
verFaeroeIsl	47	Faeroe Islands
verIran	48	Iran
verRussia	49	Russia
verIreland	50	Ireland
verKorea	51	Korea
verChina	52	People's Republic of China
verTaiwan	53	Taiwan
verThailand	54	Thailand
minCountry		The lowest defined region code (for range-checking); currently = verUS
maxCountry		The highest defined region code (for range-checking); currently = verThailand

Token Codes

The following constants define the types of tokens recognized by the IntlTokenize function and specified in the field theToken of the token record (type TokenRec):

Constant	Value	Explanation
delimPad	–2	Delimiter pad (special code)
tokenEmpty	–1	Empty flag
tokenUnknown	0	Has no existing token type
tokenWhite	1	Whitespace character
tokenLeftLit	2	Opening literal marker

Constant	Value	Explanation (continued)
tokenRightLit	3	Closing literal marker
tokenAlpha	4	Alphabetic
tokenNumeric	5	Numeric
tokenNewLine	6	New line
tokenLeftComment	7	Opening comment marker
tokenRightComment	8	Closing comment marker
tokenLiteral	9	Literal
tokenEscape	10	Escape character
tokenAltNum	11	Alternate number (such as at $B0–$B9)
tokenRealNum	12	Real number
tokenAltReal	13	Alternate real number
tokenReserve1	14	(reserved 1)
tokenReserve2	15	(reserved 2)
tokenLeftParen	16	Opening parenthesis
tokenLeftBracket	18	Opening square bracket
tokenRightBracket	19	Closing square bracket
tokenLeftCurly	20	Opening curly bracket
tokenRightCurly	21	Closing curly bracket
tokenLeftEnclose	22	Opening European double quote
tokenRightEnclose	23	Closing European double quote
tokenPlus	24	Plus
tokenMinus	25	Minus
tokenAsterisk	26	Times/multiply
tokenDivide	27	Divide
tokenSlash	29	Slash
tokenBackSlash	30	Backslash
tokenLess	31	Less than
tokenGreat	32	Greater than
tokenEqual	33	Equal
tokenLessEqual2	34	Less than or equal to (2 symbols)
tokenLessEqual1	35	Less than or equal to (1 symbol)
tokenGreatEqual2	36	Greater than or equal to (2 symbols)
tokenGreatEqual1	37	Greater than or equal to (1 symbol)
token2Equal	38	Double equal
tokenColonEqual	39	Colon equal

continued

Constant	Value	Explanation (continued)
tokenNotEqual	40	Not equal
tokenLessGreat	41	Less/greater (not equal in Pascal)
tokenExclamEqual	42	Exclamation equal (not equal in C)
tokenExclam	43	Exclamation point
tokenTilde	44	Centered tilde
tokenComma	45	Comma
tokenPeriod	46	Period
tokenLeft2Quote	47	Opening double quote
tokenRight2Quote	48	Closing double quote
tokenLeft1Quote	49	Opening single quote
tokenRight1Quote	50	Closing single quote
token2Quote	51	Double quote
token1Quote	52	Single quote
tokenSemicolon	53	Semicolon
tokenPercent	54	Percent
tokenCaret	55	Caret
tokenUnderline	56	Underline
tokenAmpersand	57	Ampersand
tokenAtSign	58	At sign
tokenBar	59	Vertical bar
tokenQuestion	60	Question mark
tokenPi	61	Pi
tokenRoot	62	Square root
tokenSigma	63	Capital sigma
tokenIntegral	64	Integral
tokenMicro	65	Micro
tokenCapPi	66	Capital pi
tokenInfinity	67	Infinity
tokenColon	68	Colon
tokenHash	69	Pound sign (U.S. weight)
tokenDollar	70	Dollar sign
tokenNoBreakSpace	71	Nonbreaking space
tokenFraction	72	Fraction
tokenIntlCurrency	73	International currency
tokenLeftSingGuillemet	74	Opening single guillemet
tokenRightSingGuillemet	75	Closing single guillemet

Constant	Value	Explanation (continued)
tokenPerThousand	76	Per thousands
tokenEllipsis	77	Ellipsis character
tokenCenterDot	78	Center dot

Selectors for Script Manager Variables

This section lists and describes the selector constants for accessing the Script Manager variables through calls to the `GetScriptManagerVariable` and `SetScriptManagerVariable` functions. In every case the variable parameter passed to or from the function is a long integer (4 bytes); the column "Size of variable" indicates how many of the 4 bytes are necessary to hold the input or return value for that variable. If fewer than 4 bytes are needed, the low byte or low word contains the information.

Descriptions of all the variables accessed by these constants follow the list.

Selector constant	Value	Size of variable (bytes)
smVersion	0	2
smMunged	2	2
smEnabled	4	1
smBidirect	6	1
smFontForce	8	1
smIntlForce	10	1
smForced	12	1
smDefault	14	1
smPrint	16	4
smSysScript	18	2
smLastScript	20	2
smKeyScript	22	2
smSysRef	24	2
smKeyCache	26	4
smKeySwap	28	4
smGenFlags	30	4
smOverride	32	4
smCharPortion	34	2
smDoubleByte	36	1
smKCHRCache	38	4
smRegionCode	40	2
smKeyDisableState	42	1

Selector constant	Variable description
`smVersion`	The Script Manager version number. This variable has the same format as the version number obtained from calling the `Gestalt` function with the Gestalt selector `gestaltScriptMgrVersion`. The high-order byte contains the major version number, and the low-order byte contains the minor version number.
`smMunged`	The modification count for Script Manager variables. At startup, `smMunged` is initialized to 0, and it is incremented when the `KeyScript` procedure changes the current keyboard script and updates the variables accessed via `smKeyScript` and `smLastScript`. The `smMunged` selector is also incremented when the `SetScriptManagerVariable` function is used to change a Script Manager variable. You can check this variable at any time to see whether any of your own data structures that may depend on Script Manager variables need to be updated.
`smEnabled`	The script count; the number of currently enabled script systems. At startup time, the Script Manager initializes the script count to 0, then increments it for each installed and enabled script system (including Roman). You can use `smEnabled` to determine whether more than one script system is installed—that is, whether your application needs to handle non-Roman text. **IMPORTANT** Never call `SetScriptManagerVariable` with the `smEnabled` selector. It could result in inconsistency with other script system values. ▲
`smBidirect`	The bidirectional flag, which indicates when at least one bidirectional script system is enabled. This flag is set to `TRUE` ($FF) if the Arabic or Hebrew script system is enabled.
`smFontForce`	The font force flag. At startup, the Script Manager sets its value from the system script's international configuration (`'itlc'`) resource. The flag returns 0 for `FALSE` and $FF for `TRUE`. If the system script is non-Roman, the font force flag controls whether a font with ID in the Roman script range is interpreted as belonging to the Roman script or to the system script. See "Using the Font Force Flag" on page 6-24. **IMPORTANT** When you call `SetScriptManagerVariable` with the `smFontForce` selector, be sure to pass only the value 0 or $FF, or a later call to `GetScriptManagerVariable` may return an unrecognized value. ▲
`smIntlForce`	The international resources selection flag. At startup, the Script Manager sets its value from the system script's international configuration (`'itlc'`) resource. The flag returns 0 for `FALSE` and `$FF` for `TRUE`. This flag controls whether international resources of the font script or the system script are used for string manipulation. See "Using the International Resources Selection Flag" on page 6-25.

IMPORTANT

When you call `SetScriptManagerVariable` with the `smIntlForce` selector, be sure to pass only the value 0 or $FF, or a later call to `GetScriptManagerVariable` may return an unrecognized value. ▲

`smForced` The script-forced result flag. If the current script has been forced to the system script, this flag is set to `TRUE`. Use the `smForced` selector to obtain reports of the actions of the `FontScript`, `FontToScript`, and `IntlScript` functions. This variable is for information only; never set its value with `SetScriptManagerVariable`.

`smDefault` The script-defaulted result flag. If the script system corresponding to a specified font is not available, this flag is set to `TRUE`. Use this selector to obtain reports of the actions of the `FontScript`, `FontToScript`, and `IntlScript` functions. This variable is for information only; never set its value with `SetScriptManagerVariable`.

`smPrint` The print action routine vector, set up by the Script Manager at startup. See *Inside Macintosh: Devices* for information on the print action routine.

`smSysScript` The system script code. At startup, the Script Manager initializes this variable from the system script's international configuration (`'itlc'`) resource. This variable is for information only; never set its value with `SetScriptManagerVariable`. Constants for all defined script codes are listed on page 6-52.

`smLastScript` The previously used keyboard script. When you change keyboard scripts with the `KeyScript` procedure, the Script Manager moves the old value of `smKeyScript` into `smLastScript`. `KeyScript` can also swap the current keyboard script with the previous keyboard script, in which case the contents of `smLastScript` and `smKeyScript` are swapped. Constants for all defined script codes are listed on page 6-52. Never set the value of this variable with `SetScriptManagerVariable`.

`smKeyScript` The current keyboard script. The `KeyScript` procedure tests and updates this variable. When you change keyboard scripts with the `KeyScript` procedure, the Script Manager moves the old value of `smKeyScript` into `smLastScript`. `KeyScript` can also swap the current keyboard script with the previous keyboard script, in which case the contents of `smLastScript` and `smKeyScript` are swapped. The Script Manager also uses this variable to get the proper keyboard icon and to retrieve the proper keyboard-layout (`'KCHR'`) resource. Constants for all defined script codes are listed on page 6-52. Never set the value of this variable directly with `SetScriptManagerVariable`; call `KeyScript` to change keyboard scripts.

`smSysRef` The System Folder volume reference number. Its value is initialized from the system global variable `BootDrive` at startup.

`smKeyCache` An obsolete variable. This variable at one time held a pointer to the keyboard cache. The value it provided was not correct and should not be used.

`smKeySwap` A handle to the keyboard-swap (`'KSWP'`) resource. The Script Manager initializes the handle at startup. The keyboard-swap resource controls the key combinations with which the user can invoke various actions with the `KeyScript` procedure, such as switching among script systems. This resource is described in the appendix "Keyboard Resources" in this book.

`smGenFlags` The general flags used by the Script Manager. The Script Manager general flags is a long word value; its high-order byte is set from the flags byte in the system script's international configuration (`'itlc'`) resource. The following constants are available to designate bits in the variable accessed through `smGenFlags`:

Constant	Value	Explanation
`smfNameTagEnab`	29	(reserved for internal use)
`smfDualCaret`	30	Use dual caret for mixed-directional text.
`smfShowIcon`	31	Show keyboard menu even if only one keyboard layout or one script (Roman) is available. (This bit is checked only at system startup.)

`smOverride` The script override flags. At present, these flags are not set or used by the Script Manager. They are, however, reserved for future use.

`smCharPortion` A value used by script systems to allocate intercharacter and interword spacing when justifying text. It denotes the weight allocated to intercharacter space versus interword space. The value of this variable is initialized to 10 percent by the Script Manager, although it currently has no effect on text of the Roman script system. The variable is in 4.12 fixed-point format, which is a 16-bit signed number with 4 bits of integer and 12 bits of fraction. (In that format, 10 percent has the hexadecimal value $0199.)

`smDoubleByte` The 2-byte flag, a Boolean value that is `TRUE` if at least one 2-byte script system is enabled.

`smKCHRCache` A pointer to the cache that stores a copy of the current keyboard-layout (`'KCHR'`) resource. The keyboard-layout resource is described in the appendix "Keyboard Resources" in this book.

`smRegionCode` The region code for this localized version of system software, obtained from the system script's international configuration (`'itlc'`) resource. This variable identifies the localized version of the system script. Constants for all defined region codes are listed starting on page 6-57.

`smKeyDisableState`
The current disable state for keyboards. The Script Manager disables some keyboard scripts or keyboard switching when text input must be restricted to certain script systems or when script systems are being moved into or out of the System file.

See "Making Keyboard Settings" beginning on page 6-17. These are the possible values for the variable accessed through `smKeyDisableState`:

Value	Explanation
0	All keyboards are enabled, switching is enabled
1	Keyboard switching is disabled
$FF	Keyboards for all non-Roman secondary scripts are disabled

The script management system maintains the keyboard disable state separately for each application. Never set the value of this variable directly with `SetScriptManagerVariable`; call `KeyScript` to change the keyboard disable state for your application.

Selectors for Script Variables

This section lists and describes the selector constants for accessing script variables through calls to the `GetScriptVariable` and `SetScriptVariable` functions. In every case the variable parameter passed to or from the function is a long integer (4 bytes); the column "Size of variable" indicates how many of the 4 bytes are necessary to hold the input or return value for that variable. If fewer than 4 bytes are needed, the low byte or low word contains the information.

In many cases the value of a script variable is taken from the script system's international bundle (`'itlb'`) resource. See the appendix "International Resources" for a description of the international bundle resource.

Descriptions of all the variables accessed by these constants follow the list.

Selector constant	Value	Size of variable (bytes)
`smScriptVersion`	0	2
`smScriptMunged`	2	2
`smScriptEnabled`	4	1
`smScriptRight`	6	1
`smScriptJust`	8	1
`smScriptRedraw`	10	1
`smScriptSysFond`	12	2
`smScriptAppFond`	14	2
`smScriptNumber`	16	2
`smScriptDate`	18	2
`smScriptSort`	20	2
`smScriptFlags`	22	2

continued

Selector constant	Value	Size of variable (bytes) (continued)
smScriptToken	24	2
smScriptEncoding	26	2
smScriptLang	28	2
smScriptNumDate	30	2
smScriptKeys	32	2
smScriptIcon	34	2
smScriptPrint	36	4
smScriptTrap	38	4
smScriptCreator	40	4
smScriptFile	42	4
smScriptName	44	4
smScriptMonoFondSize	78	4
smScriptPrefFondSize	80	4
smScriptSmallFondSize	82	4
smScriptSysFondSize	84	4
smScriptAppFondSize	86	4
smScriptHelpFondSize	88	4
smScriptValidStyles	90	1
smScriptAliasStyle	92	1

Selector constant **Variable description**

`smScriptVersion`

The script system's version number. When the Script Manager loads the script system, the script system puts its current version number into this variable. The high-order byte contains the major version number, and the low-order byte contains the minor version number.

`smScriptMunged`

The modification count for this script system's script variables. The Script Manager increments the variable accessed by the `smScriptMunged` selector each time the `SetScriptVariable` function is called for this script system. You can check this variable at any time to see whether any of your own data structures that depend on this script system's script variables need to be updated.

`smScriptEnabled`

The script-enabled flag, a Boolean value that indicates whether the script has been enabled. It is set to $FF when enabled and to 0 when not enabled. Note that this variable is not equivalent to the Script Manager variable accessed by the `smEnabled` selector, which is a count of the total number of enabled script systems.

`smScriptRight` The right-to-left flag, a Boolean value that indicates whether the primary line direction for text in this script is right-to-left or left-to-right. It is set to $FF for right-to-left text (used in Arabic and Hebrew script systems) and to 0 for left-to-right (used in Roman and other script systems).

`smScriptJust` The script alignment flag, a byte that specifies the default alignment for text in this script system. It is set to $FF for right alignment (common for Arabic and Hebrew), and it is set to 0 for left alignment (common for Roman and other script systems). This flag usually has the same value as the `smScriptRight` flag.

`smScriptRedraw`

The script-redraw flag, a byte that provides redrawing recommendations for text of this script system. It describes how much of a line should be redrawn when a user adds, inserts, or deletes text. It is set to 0 when only a character should be redrawn (used by the Roman script system), to 1 when an entire word should be redrawn (used by the Japanese script system), and to –1 when the entire line should be redrawn (used by the Arabic and Hebrew script systems). The following constants are available for the script-redraw flag:

Constant	Value	Explanation
`smRedrawChar`	0	Redraw character only
`smRedrawWord`	1	Redraw entire word
`smRedrawLine`	–1	Redraw entire line

`smScriptSysFond`

The preferred system font, the font family ID of the system font preferred for this script. In the Roman script system, this variable specifies Chicago font, whose font family ID is 0 if Roman is the system script. The preferred system font in the Japanese script system is 16384, the font family ID for Osaka.

This variable holds similar information to the variable accessed through the `smScriptSysFondSize` selector. However, changing the value of this variable has no effect on the value accessed through `smScriptSysFondSize`.

Note

Remember that in all localized versions of system software the special value of 0 is remapped to the system font ID. Thus, if an application running under Japanese system software specifies a font family ID of 0 in a routine or in the `txFont` field of the current graphics port, Osaka will be used. However, the variable accessed by `smScriptSysFond` will still show the true ID for Osaka (16384). ◆

`smScriptAppFond`

The preferred application font; the font family ID of the application font preferred for this script. In the Roman script system, the value of this variable is the font family ID for Geneva.

This variable holds similar information to the variable accessed through the `smScriptAppFondSize` selector. However, changing the value of this variable has no effect on the value accessed through `smScriptAppFondSize`.

Note

Remember that in all localized versions of system software the special value of 1 is remapped to the application font ID. For example, if an application running under Arabic system software specifies a font family ID of 1 in a routine, Nadeem will be used. However, the variable accessed by `smScriptSysFond` will still show the true ID for Nadeem (17926). ◆

`smScriptNumber` The resource ID of the script's numeric-format (`'itl0'`) resource. The numeric-format resource includes formatting information for the correct display of numbers, times, and short dates. The value of this variable is initialized from the script system's international bundle resource. See the appendix "International Resources" for a description of the numeric-format resource.

`smScriptDate` The resource ID of the script's long-date-format (`'itl1'`) resource. The long-date-format resource includes formatting information for the correct display of long dates (dates that include month or day names). The value of this variable is initialized from the script system's international bundle resource. See the appendix "International Resources" for a description of the long-date-format resource.

`smScriptSort` The resource ID of the script's string-manipulation (`'itl2'`) resource. The string-manipulation resource contains routines for sorting and tables for word selection, line breaks, character types, and case conversion of text. The value of this variable is initialized from the script system's international bundle resource. See the appendix "International Resources" for a description of the string-manipulation resource.

`smScriptFlags` The script flags word, which contains bit flags specifying attributes of the script. The value of this variable is initialized from the script system's international bundle resource. The following constants are available for examining attributes in the script flags word. Bits above 8 are nonstatic, meaning that they may change during program execution. (Note that the constant values represent bit numbers in the flags word, not masks.)

Constant	Value	Explanation
`smsfIntellCP`	0	Can support intelligent cut and paste (uses spaces as word delimiters)
`smsfSingByte`	1	Has only 1-byte characters
`smsfNatCase`	2	Has both uppercase and lowercase native characters
`smsfContext`	3	Is contextual

Constant	Value	Explanation (continued)
smsfNoForceFont	4	Does not support font forcing (ignores the font force flag)
smsfB0Digits	5	Has alternate digits at $B0–$B9; Arabic and Hebrew, for example, have their native numeric forms at this location in their character sets
smsfAutoInit	6	Is intialized by the Script Manager; 1-byte simple script systems can set this bit to avoid having to initialize themselves
smsfUnivExt	7	Uses the WorldScript I extension
smsfSynchUnstyledTE	8	Synchronizes keyboard with font for monostyled TextEdit
smsfForms	13	Use contextual forms if this bit is set; do not use them if it is cleared
smsfLigatures	14	Use contextual ligatures if this bit is set; do not use them if it is cleared
smsfReverse	15	Reverse right-to-left text to draw it in (left-to-right) display order if this bit is set; do not reorder text if this bit is cleared

The `smsfIntellCP` flag is set if this script system uses spaces as word delimiters. In such a script system it is possible to implement intelligent cut and paste, in which extra spaces are removed when a word is cut from text, and any needed spaces are added when a word is pasted into text. *Macintosh Human Interface Guidelines* recommends that you implement intelligent cut and paste in script systems that support it.

If you use the `CharToPixel` function to determine text widths, such as for line breaking, you need to clear the `smsfReverse` bit first. For more information, see the chapter "QuickDraw Text" in this book.

`smScriptToken` The resource ID of the script's tokens (`'itl4'`) resource. The tokens resource contains information for tokenizing and number formatting. The value of this variable is initialized from the script system's international bundle resource. See the appendix "International Resources" in this book for a description of the tokens resource.

`smScriptEncoding`
The resource ID of the script's (optional) encoding/rendering (`'itl5'`) resource. For 1-byte scripts, the encoding/rendering resource specifies text-rendering behavior; for 2-byte scripts, it specifies character-encoding information. The value of this variable is taken from the script system's international bundle resource. See the appendix "International Resources" for a description of the encoding/rendering resource.

`smScriptLang` The language code for this version of the script. A language is a specialized variation of a specific script system. Constants for all defined language codes are listed on page 6-54. The value of this variable is initialized from the script system's international bundle resource.

`smScriptNumDate`
The numeral code and calendar code for the script. The numeral code specifies the kind of numerals the script uses, and is in the high-order byte of the word; the calendar code specifies the type of calendar it uses and is in the low-order byte of the word. The value of this variable is initialized from the script system's international bundle resource. It may be changed during execution when the user selects, for example, a new calendar from a script system's control panel.

The following numeral-code constants are available for specifying numerals. Note that they are bit numbers, not masks:

Constant	Value	Explanation
`intWestern`	0	Western numerals
`intArabic`	1	Native Arabic numerals
`intRoman`	2	Roman numerals
`intJapanese`	3	Japanese numerals
`intEuropean`	4	European numerals
`intOutputMask`	$8000	Output mask

The following calendar-code constants are available for specifying calendars. Note that they are bit numbers, not masks:

Constant	Value	Explanation
`calGregorian`	0	Gregorian calendar
`calArabicCivil`	1	Arabic civil calendar
`calArabicLunar`	2	Arabic lunar calendar
`calJapanese`	3	Japanese calendar
`calJewish`	4	Jewish calendar
`calCoptic`	5	Coptic calendar
`calPersian`	6	Persian calendar

`smScriptKeys` The resource ID of the script's current keyboard-layout (`'KCHR'`) resource. The keyboard-layout resource is used to map virtual key codes into the correct character codes for the script; it is described in the appendix "Keyboard Resources" in this book. The value of this variable is initialized from the script system's international bundle resource. It is updated when the user selects a new keyboard layout, or when the application calls the `KeyScript` procedure. You can force a particular keyboard layout to be used with your application by setting the value of this variable and then calling `KeyScript`.

`smScriptIcon` The resource ID of the script's keyboard icon family (resource types `'kcs#'`, `'kcs4'`, and `'kcs8'`). The keyboard icon family consists of the keyboard icons displayed in the keyboard menu; it is described in the appendix "Keyboard Resources" in this book. The value of this variable is initialized from the script system's international bundle resource. Note that, unlike `smScriptKeys`, the value of this variable is *not* automatically updated when the keyboard layout changes. (System software assumes that the icon family has an identical ID to the keyboard-layout resource, and usually ignores this variable.)

`smScriptPrint` The print action routine vector, set up by the script system (or by the Script Manager if the `smsfAutoInit` bit is set) when the script is initialized. See *Inside Macintosh: Devices* for information on the print action routine.

`smScriptTrap` A pointer to the script's script-record dispatch routine (for internal use only).

`smScriptCreator`
The 4-character creator type for the script system's file, that is, the file containing the script system. For the Roman script system, it is `'ZSYS'`, for WorldScript I it is `'univ'`, and for World Script II it is `'doub'`.

`smScriptFile` A pointer to the Pascal string that contains the name of the script system's file, that is, the file containing the script system. For the Roman script system, the string is `'System'`.

`smScriptName` A pointer to a Pascal string that contains the script system's name. For the Roman script system and 1-byte simple script systems, the string is `'Roman'`. For 1-byte complex script systems, this name is taken from the encoding/rendering (`'itl5'`) resource. For 2-byte script systems, it is taken from the WorldScript II extension and is `'WorldScript II'`.

`smScriptMonoFondSize`
The default font family ID and size (in points) for monospaced text. The ID is stored in the high-order word, and the size is stored in the low-order word. The value of this variable is taken from the script system's international bundle resource. Note that not all script systems have a monospaced font.

`smScriptPrefFondSize`
Currently not used.

`smScriptSmallFondSize`
The default font family ID and size (in points) for small text, generally the smallest font and size combination that is legible on screen. The ID is stored in the high-order word, and the size is stored in the low-order word. Sizes are important; for example, a 9-point font may be too small in Chinese. The value of this variable is taken from the script system's international bundle resource.

`smScriptSysFondSize`
The default font family ID and size (in points) for this script system's preferred system font. The ID is stored in the high-order word, and the size is stored in the low-order word. The value of this variable is taken from the script system's international bundle resource.

This variable holds similar information to the variable accessed through the `smScriptSysFond` selector. If you neeed font family ID only and don't want size information, it is simpler to use `smScriptSysFond`. Note, however, that changing the value of this variable has no effect on the value accessed through `smScriptSysFond.`

`smScriptAppFondSize`
The default font family ID and size (in points) for this script system's preferred application font. The ID is stored in the high-order word, and the size is stored in the low-order word. The value of this variable is taken from the script system's international bundle resource.

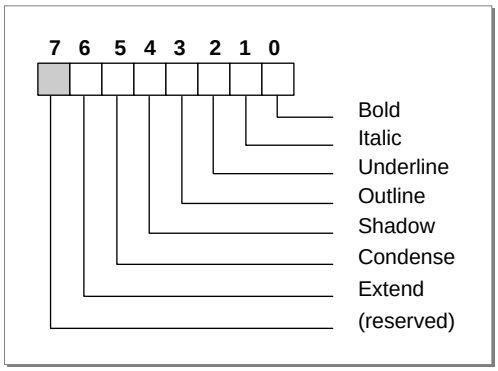
This variable holds similar information to the variable accessed through the `smScriptAppFond` selector. If you neeed font family ID only and don't want size information, it is simpler to use `smScriptAppFond`. Note, however, that changing the value of this variable has no effect on the value accessed through `smScriptAppFond`.

`smScriptHelpFondSize`
The default font family ID and size (in points) for Balloon Help. The ID is stored in the high-order word, and the size is stored in the low-order word. Sizes are important; for example, a 9-point font may be too small in Chinese. The value of this variable is taken from the script system's international bundle resource.

`smScriptValidStyles`
The set of all valid styles for the script. For example, the Extended style is not valid in the Arabic script. When the `GetScriptVariable` function is called with the `smScriptValidStyles` selector, the low-order byte of the returned value is a style code that includes all of the valid styles for

the script (that is, the bit corresponding to each QuickDraw style is set if that style is valid for the specified script). See Figure 6-7. The value of this variable is taken from the script system's international bundle resource.

Figure 6-7 Style code format



`smScriptAliasStyle`

The style to use for indicating aliases. When the `GetScriptVariable` function is called with `smScriptAliasStyle`, the low-order byte of the returned value is the style code (see Figure 6-7) that should be used in that script for indicating alias names (for example, in the Roman script system, alias names are indicated in italics). The value of this variable is taken from the script system's international bundle resource.

Note

Some script systems, such as Arabic and Hebrew, have private script-system selectors that are unique to those scripts. Those private selectors are negative, whereas selectors that extend across script systems are positive. ◆

Data Structures

This section presents the following types and data structures used by the Script Manager: the token block record and the token record. Other data type definitions are in the section "Summary of the Script Manager" beginning on page 6-107.

The Script Manager also makes use of many of the types and data structures defined in the appendix "International Resources" in this book.

Token Block Record

The token block record, of data type `TokenBlock`, is a parameter block used to pass information to the `IntlTokenize` function and to retrieve results from it.

```
TYPE
   TokenBlock =
   RECORD
      source:        Ptr;          {pointer to source text to be tokenized}
      sourceLength:  LongInt;      {length of source text in bytes}
      tokenList:     Ptr;          {pointer to array of token records}
      tokenLength:   LongInt;      {maximum size of TokenList}
      tokenCount:    LongInt;      {number of tokens currently in TokenList}
      stringList:    Ptr;          {pointer to list of token strings}
      stringLength:  LongInt;      {length available for string list}
      stringCount:   LongInt;      {current length of string list}
      doString:      Boolean;      {make strings & put into StringList?}
      doAppend:      Boolean;      {append to--not replace--TokenList?}
      doAlphanumeric:Boolean;      {identifiers may include numerics?}
      doNest:        Boolean;      {do comments nest?}
      leftDelims:    ARRAY [0..1] OF TokenType;
                                   {opening delimiters for literals}
      rightDelims:   ARRAY [0..1] OF TokenType;
                                   {closing delimiters for literals}
      leftComment:   ARRAY [0..3] OF TokenType;
                                   {opening delimiters for comments}
      rightComment:  ARRAY [0..3] OF TokenType;
                                   {closing delimiters for comments}
      escapeCode:    TokenType;    {escape symbol code}
      decimalCode:   TokenType;    {decimal symbol code}
      itlResource:   Handle;       {'itl4' resource of script for this text}
      reserved:      ARRAY [0..7] OF LongInt;
                                   {must be zero!}
   END;

   TokenBlockPtr = ^TokenBlock;
```

The fields in the token block record are described under the routine description for `IntlTokenize`, beginning on page 6-92.

Token Record

The token record (data type `TokenRec`) holds the results of the conversion of a sequence of characters to a token by the `IntlTokenize` function. When it analyzes text, `IntlTokenize` generates a token list, which is a sequence of token records.

```
TYPE
   TokenRec =
   RECORD
      theToken:         TokenType;  {numeric code for token}
      position:         Ptr;        {pointer to source text from }
                                    { which token was generated}
      length:           LongInt;    {length of source text from }
                                    { which token was generated}
      stringPosition:   StringPtr;  {pointer to Pascal string }
                                    { generated from token}
   END;
   TokenRecPtr = ^TokenRec;
```

The fields in the token record are described under the routine description for `IntlTokenize`, on page 6-95.

Routines

The Script Manager routines documented in this section allow you to

- control the system direction
- access Script Manager variables
- access script variables
- control the keyboard and keyboard script
- determine script codes
- obtain character-type information
- directly access a script system's international resources
- tokenize text
- transliterate text
- replace the default routines for a 1-byte complex script system

Throughout these routine descriptions, unless otherwise noted, the Script Manager expects that

- there is a buffer containing text characters only; font and style information are stored separately
- the storage order of the characters—the order in which character codes are stored in memory—is their logical order, the order in which they would most naturally be entered from the keyboard
- all offsets within text buffers are zero-based and specified in bytes, not characters
- a valid graphics port exists, and the font of the port is set correctly; all text-related fields in the graphics port record reflect the characteristics of the text being manipulated

Assembly-language note
You can invoke each of the Script Manager routines that uses the `_ScriptUtil` trap with a macro that has the same name as the routine, preceded by an underscore. See "Summary of the Script Manager" at the end of this chapter for a list of the routines that use the `_ScriptUtil` trap. ◆

Checking and Setting the System Direction

The `GetSysDirection` routine returns the value of `SysDirection`, the global variable that represents the system direction. A value of 0 for `SysDirection` means that the primary line direction is left-to-right; a value of –1 means that the primary line direction is right-to-left. The value of `SysDirection` is initialized from the system's international configuration resource, and may be controlled by the user. Your application can use the `SetSysDirection` procedure to change `SysDirection` while drawing, but should restore it when appropriate (such as when your application becomes inactive).

GetSysDirection

The `GetSysDirection` function returns the current value of `SysDirection`, the global variable that determines the system direction (primary line direction).

```
FUNCTION GetSysDirection: Integer;
```

DESCRIPTION

There are two possible return values from `GetSysDirection`:

0 = left-to-right line direction

–1 = $FFFF = right-to-left line direction

SetSysDirection

The `SetSysDirection` procedure sets the value of `SysDirection`, the global variable that determines the system direction (primary line direction).

```
PROCEDURE SetSysDirection (newDirection: Integer);
```

`newDirection`
The desired value for `SysDirection`.

DESCRIPTION

There are two valid input values for `newDirection`:

0 = left-to-right line direction

–1 = $FFFF = right-to-left line direction

Checking and Setting Script Manager Variables

The Script Manager maintains a set of variables that control general settings of the text environment, including the identity of the system script and the keyboard script, and the settings of the font force flag and the international resources selection flag.

You may want access to the Script Manager variables in order to understand the current environment or to modify it. The `GetScriptManagerVariable` function retrieves the values of the Script Manager variables, and the `SetScriptManagerVariable` function sets their values. (The variables themselves are private and you cannot access them directly.) When you call either routine, you use a selector to describe the variable that interests you. The integer constants for all defined `GetScriptManagerVariable`/`SetScriptManagerVariable` selectors are described beginning on page 6-61.

GetScriptManagerVariable

The `GetScriptManagerVariable` function retrieves the value of the specified Script Manager variable.

```
FUNCTION GetScriptManagerVariable (selector: Integer): LongInt;
```

`selector` A value that specifies a particular Script Manager variable.

DESCRIPTION

Although `GetScriptManagerVariable` always returns a long integer, the actual value may be a long integer, standard integer, or signed byte. If the value is not a long integer, it is stored in the low-order word or byte of the long integer returned by `GetScriptManagerVariable`; the remaining bytes are set to 0.

The `GetScriptManagerVariable` function returns 0 if the selector is invalid.

Note
For some valid selectors, 0 may also be a valid return value. For example, when you call `GetScriptManagerVariable` with a selector value of `smRegionCode` on a version of Macintosh system software that has been localized for the United States, it returns 0. ◆

To specify the Script Manager variable whose value you need, use one of the selector constants listed on page 6-61.

SetScriptManagerVariable

The `SetScriptManagerVariable` function sets the specified Script Manager variable to the value of the input parameter.

```
FUNCTION SetScriptManagerVariable (selector: Integer;
                                   param: LongInt): OSErr;
```

`selector` A value that specifies a particular Script Manager variable.

`param` The new value for the specified Script Manager variable.

DESCRIPTION

The actual values to be assigned may be long integers, standard integers, or signed bytes. If the value is other than a long integer, you must store it in the low-order word or byte of the `param` parameter and set the unused bytes to 0.

The `SetScriptManagerVariable` function returns the value `smBadVerb` if the selector is not valid. Otherwise, it returns 0 (`noErr`).

To specify the Script Manager variable whose value you wish to change, use one of the selector constants listed on page 6-61.

RESULT CODES

`noErr`	0	No error
`smBadVerb`	–1	Invalid selector passed to the routine

Checking and Setting Script Variables

Each enabled script system maintains a set of variables that control the current settings of that script system, including the ID numbers of its international resources, its preferred fonts and font sizes, and its primary line direction.

You may want access to the script variables in order to conform to the script's current settings or to modify them. The `GetScriptVariable` function retrieves the values of the script variables, and the `SetScriptVariable` function sets their values. (The variables themselves are private and you cannot access them directly.) When you call either routine, you use a selector to describe the variable that interests you. The integer constants for all defined `GetScriptVariable`/`SetScriptVariable` selectors are described on page 6-65.

GetScriptVariable

The `GetScriptVariable` function retrieves the value of the specified script variable from the specified script system.

```
FUNCTION GetScriptVariable (script: ScriptCode;
                            selector: Integer): LongInt;
```

`script` A value that specifies the script system whose variable you are accessing.

`selector` A value that specifies a particular script variable.

DESCRIPTION

Although `GetScriptVariable` always returns a long integer, the actual value may be a long integer, standard integer, or signed byte. If the value is not a long integer, it is stored in the low-order word or byte of the long integer returned by `GetScriptVariable`; the remaining bytes are set to 0.

Valid selector values are defined by each script system. `GetScriptVariable` returns 0 if the selector value is invalid or if the specified script system is not installed.

Note
For some valid selectors, 0 may also be a valid return value. For example, calling `GetScriptVariable` with a selector of `smScriptLang` on a version of Macintosh system software that has been localized for the United States returns 0. ◆

To specify the script variable whose value you need, use one of the selector constants listed on page 6-65. To specify the script system, use one of the script-code constants listed on page 6-52.

SetScriptVariable

The `SetScriptVariable` function sets the specified script variable for the specifed script system to the value of the input parameter.

```
FUNCTION SetScriptVariable (script: ScriptCode; selector: Integer;
                            param: LongInt): OSErr;
```

`script` A value that specifies the script system whose variable you are setting.

`selector` A value that specifies a particular script variable.

`param` The new value for the specified script variable.

DESCRIPTION

The actual values to be assigned may be long integers, standard integers, or signed bytes. If the value is not a long integer, you must store it in the low-order word or byte of the `param` parameter and set the unused bytes to 0.

The `SetScriptVariable` function returns the value `smBadVerb` if the selector is not valid, and `smBadScript` if the script is invalid. Otherwise, it returns 0 (`noErr`).

To specify the script variable whose value you wish to change, use one of the selector constants listed on page 6-65. To specify the script system, use one of the script-code constants listed on page 6-52.

RESULT CODES

`noErr`	0	No error
`smBadVerb`	–1	Invalid selector passed to the routine
`smBadScript`	–2	Invalid script code passed to the routine

Making Keyboard Settings

The Script Manager provides the `KeyScript` procedure to let you specify the current keyboard script (the script system used for keyboard input), keyboard layout (the mapping of keys to characters), or input method (a facility for entering 2-byte characters), and to make various settings related to text input.

For the purposes of `KeyScript`, *keyboard layout* means a keyboard-layout (`'KCHR'`) resource, plus optionally a key-remap (`'itlk'`) resource. To change keyboard layouts means to change the current keyboard-layout resource.

KeyScript

The `KeyScript` procedure uses the supplied value to change the keyboard script, to change the keyboard layout or input method within the current keyboard script, or to make a setting related to text input. If the Keyboard menu is displayed, `KeyScript` also updates the Keyboard menu.

```
PROCEDURE KeyScript (code: Integer);
```

`code` If 0 or positive, directly specifies a script system (that is, it is read as a script code). Negative values have special meanings.

DESCRIPTION

The `Keyscript` procedure makes the change based on the selector with which it is called. If more than one script system is enabled or if the `smfShowIcon` bit flag is set in the Script Manager variable accessed by the `GetScriptManagerVariable` selector

`smGenFlags`, `Keyscript` also updates the Keyboard menu by changing the icon displayed on the menu bar and placing a check beside the appropriate keyboard menu item.

The `code` parameter is a selector that can *explicitly* specify a keyboard script by script code. Script code constants are listed on page 6-52. If the selector specifies a script, then the current default keyboard layout (`'KCHR'` resource) for that script, as specified in the script's international bundle resource, becomes the current keyboard layout.

The selector can also *implicitly* specify a keyboard script (for example, the next script), a keyboard layout (for example, the previously used keyboard layout in the current script), or an input method (for example, inline input versus window-based input). It can also specify settings that enable or disable keyboard layouts and keyboard scripts, and toggle among input options or line direction. The valid constants for the `code` parameter are listed in Table 6-5 on page 6-18.

If you call `KeyScript` and explicitly specify a script system that is not available, `KeyScript` does nothing. The current keyboard script remains unchanged.

SPECIAL CONSIDERATIONS

`KeyScript` operates only on those keyboard-layout and key-remap resources that are present in the System file.

Your application's keyboard-menu setting is not maintained by the Process Manager; if the state of the keyboard menu is changed while you are switched out, the Process Manager does not restore your setting when you are switched back in. However, the Process Manager does maintain the keyboard disable state (Script Manager variable `smKeyDisableState`) for your application. See "Selectors for Script Manager Variables" beginning on page 6-61 for a description of the `smKeyDisableState` variable.

`KeyScript` may move memory; your application should not call this procedure at interrupt time.

SEE ALSO

The Process Manager is described in *Inside Macintosh: Processes*.

Determining Script Codes From Font Information

The `FontScript`, `FontToScript`, and `IntlScript` functions give you ways to determine a script code from font information. This information is subject to two control flags—the font force flag and the international resources selection flag. You can test and set these flags with the `GetScriptManagerVariable` and `SetScriptManagerVariable` selectors `smFontForce` and `smIntlForce`. For more information on the font force flag, see "Using the Font Force Flag" on page 6-24. For more information on the international resources selection flag, see "Using the International Resources Selection Flag" on page 6-25.

The routines start by initializing two result flags, the script-forced result flag and the script-defaulted result flag, to FALSE. These flags are Script Manager variables, accessed through the GetScriptManagerVariable function selectors smForced and smDefault.

FontScript

The FontScript function returns the script code for the current script. The current script is usually the font script.

```
FUNCTION FontScript: Integer;
```

DESCRIPTION

The FontScript function returns a script code. All recognized script codes and their defined constants are listed on page 6-52. FontScript returns only explicit script codes (≥ 0).

If the font of the active graphics port is Roman and the font force flag is TRUE, the script code returned is that of the system script and the script-forced result flag is set to TRUE.

If the font of the active graphics port is non-Roman, the state of the font force flag is ignored.

If the script system corresponding to the font of the active graphics port is not installed and enabled, the script code returned is that of the system script and the script-defaulted result flag is set to TRUE.

SPECIAL CONSIDERATIONS

FontScript may move memory; your application should not call this function at interrupt time.

FontToScript

The FontToScript function translates a font family ID number into its corresponding script code, if that script system is currently enabled.

```
FUNCTION FontToScript (fontNumber: Integer): Integer;
```

fontNumber
: A font family ID number.

DESCRIPTION

The `FontToScript` function returns a script code. All recognized script codes and their defined constants are listed on page 6-52. `FontToScript` returns only explicit script codes (≥ 0).

If `fontNumber` is in the Roman range and the font force flag is `TRUE`, the script code returned is that of the system script and the script-forced result flag is set to `TRUE`.

If `fontNumber` is in the non-Roman range, the state of the font force flag is ignored.

If the script system corresponding to `fontNumber` is not enabled, the script code returned is that of the system script and the script-defaulted result flag is set to `TRUE`.

SPECIAL CONSIDERATIONS

`FontToScript` may move memory; your application should not call this function at interrupt time.

IntlScript

The `IntlScript` function identifies the script system used by the Text Utilities date-formatting, time-formatting, and string-sorting routines. It also identifies the script system whose resources are returned by the Script Manager function `GetIntlResource`. It is either the font script—the script system corresponding to the current font of the active graphics port—or the system script.

```
FUNCTION IntlScript: Integer;
```

DESCRIPTION

The `IntlScript` function returns a script code. All recognized script codes and their defined constants are listed on page 6-52. `IntlScript` returns only explicit script codes (≥ 0).

If the international resources selection flag is `TRUE`, the script code returned is that of the system script.

If the identified script system is not enabled, the script code returned is that of the system script and the script-defaulted result flag is set to `TRUE`.

SPECIAL CONSIDERATIONS

`IntlScript` may move memory; your application should not call this function at interrupt time.

Analyzing Characters

This section describes the functions `CharacterByteType`, `CharacterType`, and `FillParseTable`, which give you information about a character or group of characters, specified by character code:

- The `CharacterByteType` function identifies a byte in a text buffer as a 1-byte character or as the first or second byte of a 2-byte character.
- The `CharacterType` function returns specific information about the character at a particular byte offset.
- The `FillParseTable` function fills a 256-byte table that indicates, for each possible byte value, whether it is the first byte of a 2-byte character.

The script system associated with the character you wish to examine must be enabled in order for any of these three routines to provide useful information. For example, if only the Roman script system is available and you attempt to identify a byte in a run of 2-byte characters, the `CharacterByteType` function returns 0, indicating that the byte is a 1-byte character.

1-byte script systems

For 1-byte script systems, the character-type tables reside in the string-manipulation (`'itl2'`) resource and reflect region-specific or language-specific differences in uppercase conventions. The `CharacterType` function gets the tables from the string-manipulation resource using the `GetIntlResource` function. ◆

2-byte script systems

For 2-byte script systems, the character-type tables reside in the encoding/rendering (`'itl5'`) resource, not the string-manipulation resource. Whenever you call `CharacterByteType`, `CharacterType`, or `FillParseTable`, the necessary character-set encoding information is taken from the encoding/rendering resource. You cannot use the `GetIntlResource` function to access 2-byte character-type tables directly. ◆

CharacterByteType

The `CharacterByteType` function identifies a byte in a text buffer as a 1-byte character or as the first or second byte of a 2-byte character.

```
FUNCTION CharacterByteType (textBuf: Ptr; textOffset: Integer;
                            script: ScriptCode): Integer;
```

`textBuf` A pointer to a text buffer containing the byte to be identified.

`textOffset`
The offset to the byte to be identified. Offset is measured in bytes; the first byte has an offset of 0.

`script` A value that specifies the script system of the text in the buffer. Constants for all defined script codes are listed on page 6-52. To specify the font script, pass `smCurrentScript` in this parameter.

DESCRIPTION

`CharacterByteType` returns one of three identifications: a 1-byte character, the first byte of a 2-byte character, or the second byte of a 2-byte character. The first byte of a 2-byte character—the one at the lower offset in memory—is the high-order byte; the second byte of a 2-byte character—the one at the higher offset—is the low-order byte. This is the same order in which text is processed and numbers are represented.

From byte value alone, it is not possible to distinguish the second byte of a 2-byte character from a 1-byte character. See the discussion of character encoding in the chapter "Introduction to Text on the Macintosh" in this book. `CharacterByteType` differentiates the second byte of a 2-byte character from a 1-byte character by assuming that the byte at offset 0 is the first byte of a character. With that assumption, it then sequentially identifies the size and starting position of each character in the buffer up to `textOffset`.

SPECIAL CONSIDERATIONS

If you specify `smCurrentScript` for the `script` parameter, the value returned by `CharacterByteType` can be affected by the state of the font force flag. It is unaffected by the state of the international resources selection flag.

RESULT CODES

`smFirstByte`	–1	First byte of a 2-byte character
`smSingleByte`	0	1-byte character
`smLastByte`	1	Second byte of 2-byte character

CharacterType

The `CharacterType` function returns a variety of information about the character represented by a given byte, including its type, class, orientation, direction, case, and size (in bytes).

```
FUNCTION CharacterType (textBuf: Ptr; textOffset: Integer;
                        script: ScriptCode): Integer;
```

`textBuf` A pointer to a text buffer containing the character to be examined.

`textOffset`
The offset to the location of the character to be examined. (It can be an offset to either the first or the second byte of a 2-byte character.) Offset is in bytes; the first byte of the first character has an offset of 0.

`script` A value that specifies the script system the byte belongs to. Constants for all defined script codes are listed on page 6-52. To specify the font script, pass `smCurrentScript` in this parameter.

DESCRIPTION

The `CharacterType` return value is an integer bit field that provides information about the requested character. The field has the following format:

Bit range	Name	Explanation
0–3	Type	Character types
4–7		(reserved)
8–11	Class	Character classes (= subtypes)
12	Orientation	Horizontal or vertical
13	Direction	Left or right*
14	Case	Uppercase or lowercase
15	Size	1-byte or 2-byte

* In 2-byte script systems, bit 13 indicates whether or not the character is part of the main character set (not a user-defined character).

The Script Manager defines the recognized character types, character classes, and character modifiers (bits 12–15), with constants to describe them. All of the constants are listed and described in the section "Getting Character-Type Information" beginning on page 6-28.

The Script Manager also defines a set of masks with which you can isolate each of the fields in the `CharacterType` return value. If you perform an `AND` operation with the `CharacterType` result and the mask for a particular field, you select only the bits in that field. Once you've done that, you can test the result, using the constants that represent the possible results.

The `CharacterType` field masks are the following:

Mask	Hex. value	Explanation
`smcTypeMask`	$000F	Character-type mask
`smcReserved`	$00F0	(reserved)
`smcClassMask`	$0F00	Character-class mask
`smcOrientationMask`	$1000	Character orientation (2-byte scripts)
`smcRightMask`	$2000	Writing direction (bidirectional scripts) Main character set or subset (2-byte scripts)
`smcUpperMask`	$4000	Uppercase or lowercase
`smcDoubleMask`	$8000	Size (1 or 2 bytes)

The character type of the character in question is the result of performing an `AND` operation with `smcTypeMask` and the `CharacterType` result. Constants for the defined character types are listed on page 6-28.

The character class of the character in question is the result of performing an AND operation with `smcClassMask` and the `CharacterType` result. Character classes can be considered as subtypes of character types. Constants for the defined character classes are listed on page 6-29.

The orientation of the character in question is the result of performing an AND operation with `smcOrientationMask` and the `CharacterType` result. The orientation value can be either `smCharHorizontal` or `smCharVertical`.

The direction of the character in question is the result of performing an AND operation with `smcRightMask` and the `CharacterType` result. The direction value can be either `smCharLeft` (left-to-right) or `smCharRight` (right-to-left).

The case of the character in question is the result of performing an AND operation with `smcUpperMask` and the `CharacterType` result. The case value can be either `smCharLower` or `smCharUpper`.

The size of the character in question is the result of performing an AND operation with `smcDoubleMask` and the `CharacterType` result. The size value can be either `smChar1byte` or `smChar2byte`.

Note
`CharacterType` calls `CharacterByteType` to determine whether the byte at `textOffset` is a 1-byte character or the first byte or second byte of a 2-byte character. The larger the text buffer, the longer `CharacterByteType` takes to execute. To be most efficient, place the pointer `textBuf` at the beginning of the character of interest before calling `CharacterType`. (If you want to be compatible with older versions of `CharacterType`, also set `textOffset` to 1, rather than 0, for 2-byte characters.) ◆

SPECIAL CONSIDERATIONS

`CharacterType` may move memory; your application should not call this function at interrupt time.

If you specify `smCurrentScript` for the `script` parameter, `CharacterType` always assumes that the text in the buffer belongs to the font script. It is unaffected by the state of the font force flag or the international resources selection flag.

For 1-byte script systems, the character-type tables are in the string-manipulation (`'itl2'`) resource. For 2-byte script systems, they are in the encoding/rendering (`'itl5'`) resource. If the appropriate resource does not include these tables, `CharacterType` exits without doing anything.

Some Roman fonts (for example, Symbol) substitute other characters for the standard characters in the Standard Roman character set. Since the Roman script system `CharacterType` function assumes the Standard Roman character set, it may return inappropriate results for nonstandard characters.

In versions of system software earlier than 7.0, the `textOffset` parameter to the `CharacterType` function must point to the second byte of a 2-byte character.

RESULT CODES

The complete set of `CharacterType` return values is found in the section "Getting Character-Type Information" beginning on page 6-28.

FillParseTable

The `FillParseTable` function helps your application to quickly process a buffer of mixed 1-byte and 2-byte characters. It returns a 256-byte table that distinguishes the character codes of all possible 1-byte characters from the first (high-order) byte values of all possible 2-byte characters in the specified script system.

```
FUNCTION FillParseTable (VAR table: CharByteTable;
                         script: ScriptCode): Boolean;
```

`table` A 256-byte table to be filled in by `FillParseTable`.

`script` A value that specifies the script system the parse table belongs to. Constants for all defined script codes are listed on page 6-52. To specify the font script, pass `smCurrentScript` in this parameter.

DESCRIPTION

Before calling `FillParseTable`, allocate space for a 256-byte table to pass to the function in the `table` parameter.

The information returned by `FillParseTable` is a packed array defined by the `CharByteTable` data type as follows:

```
CharByteTable = PACKED ARRAY[0..255] OF SignedByte;
```

In every script system, 2-byte characters have distinctive high-order (first) bytes that allow them to be distinguished from 1-byte characters. `FillParseTable` fills a 256-byte table, conceptually equivalent to a 1-byte character-set table, with values that indicate, byte-for-byte, whether the character-code value represented by that byte index is the first byte of a 2-byte character. An entry in the `CharByteTable` is 0 for a 1-byte character and 1 for the first byte of a 2-byte character.

If your application is processing mixed characters, it can use the table to identify the locations of the 2-byte characters as it makes a single pass through the text, rather than having to call `CharacterByteType` or `CharacterType` for each byte of the text buffer in turn. `CharacterByteType` and `CharacterType` start anew at the beginning of the text buffer each time they are called, tracking character positions up to the offset of the byte to be analyzed.

SPECIAL CONSIDERATIONS

`FillParseTable` may move memory; your application should not call this function at interrupt time.

The table defined by `CharByteTable` is not dynamic; it does not get updated when the current font changes. You need to call it separately for each script run in your text.

The return value from `FillParseTable` is always `TRUE`.

If you specify `smCurrentScript` for the `script` parameter, the value returned by `FillParseTable` can be affected by the state of the font force flag. It is unaffected by the international resources selection flag.

Directly Accessing International Resources

You can access the International resources (resource types `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, and `'itl5'`) with the `GetIntlResource` function. You can access specific tables within an international resource with the `GetIntlResourceTable` procedure. If your application provides its own `'itl2'` or `'itl4'` resources, it should call the `ClearIntlResourceCache` procedure before accessing those resources.

ClearIntlResourceCache

The `ClearIntlResourceCache` procedure clears the application's international resources cache, which contains the resource ID numbers of the string-manipulation (`'itl2'`) and tokens (`'itl4'`) resources for the current script.

```
PROCEDURE ClearIntlResourceCache;
```

DESCRIPTION

At application launch, the script management system sets up an international resources cache for the application. The cache contains the resource ID numbers of the string-manipulation and tokens resources for all enabled scripts.

If you provide your own string manipulation or tokens resource to replace the default for a particular script, call `ClearIntlResourceCache` at launch to ensure that your supplied resource is used instead of the script system's `'itl2'` or `'itl4'` resource.

The current default ID numbers for a script system's `'itl2'` and `'itl4'` resources are stored in its script variables. You can read and modify these values with the `GetScriptVariable` and `SetScriptVariable` functions using the selectors `smScriptSort` (for the `'itl2'` resource) and `smScriptToken` (for the `'itl4'` resource). Before calling `ClearIntlResourceCache`, you should set the script's default ID number to the ID of the resource that you are supplying.

If the international resources selection flag is `TRUE`, the ID numbers of your supplied resources must be in the system script range. Otherwise, the IDs must be in the range of the current script.

IMPORTANT

If you use the `SetScriptVariable` function to change the value of the `'itl2'` or `'itl4'` resource ID and then call `ClearIntlResourceCache` to flush the cache, be sure to restore the original resource ID before your application quits. ▲

SPECIAL CONSIDERATIONS

`ClearIntlResourceCache` may move memory; your application should not call this procedure at interrupt time.

GetIntlResource

The `GetIntlResource` function returns a handle to one of the following international resources: numeric-format (`'itl0'`), long-date-format (`'itl1'`), string-manipulation (`'itl2'`), tokens (`'itl4'`), or encoding/rendering (`'itl5'`). `GetIntlResource` selects the resource of the requested type for the current script.

```
FUNCTION GetIntlResource (theID: Integer) :Handle;
```

`theID` Contains an integer (0, 1, 2, 4, or 5 respectively for the `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, and `'itl5'` resources) to identify the type of the desired international resource.

DESCRIPTION

The `GetIntlResource` function returns a handle to the correct resource of the requested type. The resource returned is that of the current script, which is either the font script or the system script. See "Determining Script Codes From Font Information" on page 6-21.

If `GetIntlResource` cannot return the requested resource, it returns a `NIL` handle and sets the global variable `resErr` to the appropriate error code.

SPECIAL CONSIDERATIONS

`GetIntlResource` may move memory; your application should not call this function at interrupt time.

SEE ALSO

See the Resource Manager chapter in *Inside Macintosh: More Macintosh Toolbox* for information on `resErr` and how to get its value.

GetIntlResourceTable

The `GetIntlResourceTable` procedure gives you access to a specific word-selection, line-break, number-parts, untoken, or whitespace table from the appropriate international resource.

```
PROCEDURE GetIntlResourceTable (script: ScriptCode;
                                tableCode: Integer;
                                VAR itlHandle: Handle;
                                VAR offset: LongInt;
                                VAR length: LongInt);
```

`script` A script code, the value that specifies a particular script system. Constants for all defined script codes are listed on page 6-52.

`tableCode` A number that specifies which table is requested.

`itlHandle` Upon completion of the call, contains a handle to the string-manipulation (`'itl2'`) or tokens (`'itl4'`) resource containing the table specified in the `tableCode` parameter.

`offset` Upon completion of the call, contains the offset (in bytes) to the specified table from the beginning of the resource.

`length` Upon completion of the call, contains the size of the table (in bytes).

DESCRIPTION

When you provide a script code in the `script` parameter, and a table code in the `tableCode` parameter, `GetIntlResourceTable` returns a handle to the string-manipulation resource or tokens resource containing that table, the offset of the specified table from the beginning of the resource, and the length of the table.

If the script system whose table is requested is not available, `GetIntlResourceTable` returns a `NIL` handle.

Constants for all defined script codes are listed on page 6-52.

These are the defined constants for `tableCode`:

Constant	Value	Explanation
`smWordSelectTable`	0	Word-break table
`smWordWrapTable`	1	Line-break table
`smNumberPartsTable`	2	Number-parts table
`smUnTokenTable`	3	Untoken table
`smWhiteSpaceList`	4	Whitespace table

If you wish to manipulate the contents of the table you have requested, use the size returned in the `length` parameter to allocate a buffer, and perform a block move of the table's contents into that buffer.

SPECIAL CONSIDERATIONS

`GetIntlResourceTable` may move memory; your application should not call this procedure at interrupt time.

SEE ALSO

Block moves are described in the Memory Manager chapter of *Inside Macintosh: Memory*.

Tokenization

The Script Manager provides a way to take programming-language text in an arbitrary script system and break it into tokens: language-independent symbols that can be used as input to a parser. The `IntlTokenize` function uses information in a script system's tokens (`'itl4'`) resource to convert text to tokens that stand for names, symbols, comments, and quoted literals.

IntlTokenize

The `IntlTokenize` function allows your application to convert text into a sequence of language-independent tokens. It returns a list of tokens that correspond to the text that you pass it.

```
FUNCTION IntlTokenize (tokenParam: TokenBlockPtr): TokenResults;
```

`tokenParam`

A pointer to a token block record. The record specifies the text to be converted to tokens, the destination of the token list, a handle to the tokens (`'itl4'`) resource, and a set of options.

The token block record is a parameter block and a data structure of type `TokenBlock`, described on page 6-74. You specify input values and receive return values in as shown here:

Parameter block

→	`source`	`Ptr`	A pointer to the beginning of the source text (not a Pascal string) to be converted.
→	`sourceLength`	`LongInt`	The number of bytes in the source text.
↔	`tokenList`	`Ptr`	A pointer to a buffer you have allocated, into which the `IntlTokenize` function places the list of token records it generates.
→	`tokenLength`	`LongInt`	The maximum size of token list (in number of tokens, not bytes) that will fit into the buffer pointed to by the `tokenList` field.
↔	`tokenCount`	`LongInt`	On input: If `doAppend` = `TRUE`, must contain the correct number of tokens currently in the token list. (Ignored if `doAppend` = `FALSE`.) On output: The number of tokens currently in the token list.
↔	`stringList`	`Ptr`	If `doString` = `TRUE`, must contain a pointer to a buffer into which `IntlTokenize` can place a list of strings it generates. (Ignored if `doString` = `FALSE`.)
→	`stringLength`	`LongInt`	If `doString` = `TRUE`, must contain the size in bytes of the string list buffer pointed to by the `stringList` field. (Ignored if `doString` = `FALSE`.)
↔	`stringCount`	`LongInt`	On input: If `doString` = `TRUE` and `doAppend` = `TRUE`, must contain the correct current size in bytes of the string list. (Ignored if `doString` = `FALSE` or `doAppend` = `FALSE`.) On output: The current size in bytes of the string list. (Indeterminate if `doString` = `FALSE`.)
→	`doString`	`Boolean`	If `TRUE`, instructs `IntlTokenize` to create a Pascal string representing the contents of each token it generates. If `FALSE`, `IntlTokenize` generates a token list without an associated string list.
→	`doAppend`	`Boolean`	If `TRUE`, instructs `IntlTokenize` to append tokens and strings it generates to the current token list and string list. If `FALSE`, `IntlTokenize` writes over any previous contents of the buffer pointed to by `tokenList` and `stringList`.

→	doAlphanumeric	Boolean	If TRUE, instructs IntlTokenize to interpret numeric characters as alphabetic when mixed with alphabetic characters. If FALSE, all numeric characters are interpreted as numbers.
→	doNest	Boolean	If TRUE, instructs IntlTokenize to allow nested comments (to any depth of nesting). If FALSE, comment delimiters may not be nested within other comment delimiters.
→	leftDelims	DelimType	An array of two integers, each of which contains the token code of the symbol that may be used as an opening delimiter for a quoted literal. If only one opening delimiter is needed, the other must be specified to be delimPad.
→	rightDelims	DelimType	An array of two integers, each of which contains the token code of the symbol that may be used as the matching closing delimiter for the corresponding opening delimiter in the leftDelims field.
→	leftComment	CommentType	An array of two pairs of integers, each pair of which contains codes for the two token types that may be used as opening delimiters for comments.
→	rightComment	CommentType	An array of two pairs of integers, each pair of which contains codes for the two token types that may be used as closing delimiters for comments.
→	escapeCode	TokenType	A single integer that contains the token code for the symbol that may be an escape character within a quoted literal.
→	decimalCode	TokenType	A single integer that contains the token type of the symbol to be used for a decimal point.
→	itlResource	Handle	A handle to the tokens ('itl4') resource of the script system under which the source text was created.
→	reserved	ARRAY [0..7] OF LongInt	Must be set to 0.

DESCRIPTION

The IntlTokenize function returns a list of tokens that correspond to the input text. The token list is an array of token records (type TokenRec). Each token record describes the token generated, specifies the part of the source text it came from, and optionally provides a character string that is a normalized version of the text that generated the token.

`IntlTokenize` also returns a result code that specifies the type of error that occurred, if any.

Before calling the `IntlTokenize` function, allocate memory for and set up the following data structures:

- A token block record (data type `TokenBlock`). The token block record is a parameter block that holds both input and output parameters for the `IntlTokenize` function.
- A token list to hold the results of the tokenizing operation. To set up the token list, estimate how many tokens will be generated from your text, multiply that by the size of a token record, and allocate a memory block of that size in bytes. An upper limit to the possible number of tokens is the number of characters in the source text.
- A string list, if you want the `IntlTokenize` function to generate character strings for all the tokens. To set up the string list, multiply the estimated number of tokens by the expected average size of a string, and allocate a memory block of that size in bytes. An upper limit is twice the number of tokens plus the number of bytes in the source text.

`IntlTokenize` creates tokens based on information in the tokens (`'itl4'`) resource of the script system under which the source text was created. You must load the tokens resource and place its handle in the token block record before calling the `IntlTokenize` function.

The token block record contains both input and output values. At input, you must provide values for the fields that specify the source text location, the token list location, the size of the token list, the tokens (`'itl4'`) resource to use, and several options that affect the operation. You must set reserved locations to 0 before calling `IntlTokenize`.

On output, the token block record specifies how many tokens have been generated and the size of the string list (if you have selected the option to generate strings).

The results of the tokenizing operation are contained in the token list, an array of token records. A token record (data type `TokenRec`) consists of a token code, a pointer to a location in the source text, the length of a character sequence in the source text, and an optional pointer to a Pascal string:

```
TYPE
   TokenRec =
   RECORD
      theToken:         TokenType;  {numeric code for token}
      position:         Ptr;        {pointer to source text from }
                                    { which token was generated}
      length:           LongInt;    {length of source text from }
                                    { which token was generated}
      stringPosition:   StringPtr;  {pointer to Pascal string }
                                    { generated from token}
   END;
   TokenRecPtr = ^TokenRec;
```

Field descriptions

`theToken`	The token code that specifies the type of token (such as whitespace, opening parenthesis, alphabetic or numeric sequence) described by this token record. Constants for all defined token codes are listed on page 6-58.
`position`	A pointer to the first character in the source text that caused this particular token to be generated.
`length`	The length in bytes of the source text that caused this particular token to be generated.
`stringPosition`	If `doString` = `TRUE`, a pointer to a null-terminated Pascal string, padded if necessary so that its total number of bytes (length byte + text + null byte + padding) is even. If `doString` = `FALSE`, this field is `NIL`.

Note

The value in the length byte of the null-terminated Pascal string does not include either the terminating zero byte or the possible additional padding byte. There may be as many as two additional bytes beyond the specified length. ◆

Pascal strings are generated if the `doString` parameter in the token block record is set to `TRUE`. The string is a normalized version of the source text that generated the token; alternate digits are replaced with ASCII numerals, the decimal point is always an ASCII period, and 2-byte Roman letters are replaced with low-ASCII equivalents.

To make a series of calls to `IntlTokenize` and append the results of each call to the results of previous calls, set `doAppend` to `FALSE` and initialize `tokenCount` and `stringCount` to 0 before making the first call to `IntlTokenize`. (You can ignore `stringCount` if you set `doString` to `FALSE`.) Upon completion of the call, `tokenCount` and `stringCount` will contain the number of tokens and the length in bytes of the string list, respectively, generated by the call. On subsequent calls, set `doAppend` to `TRUE`, reset the `source` and `sourceLength` parameters (and any other parameters as appropriate) for the new source text, but maintain the output values for `tokenCount` and `stringCount` from each call as input values to the next call. At the end of your sequence of calls, the token list and string list will contain, in order, all the tokens and strings generated from the calls to `IntlTokenize`.

If you are making tokens from text that was created under more than one script system, you must load the proper tokens resource and place its handle in the token block record separately for each script run in the text, appending the results each time.

Delimiters for quoted literals are passed to `IntlTokenize` in a two-integer array:

```
TYPE DelimType = ARRAY[0..1] OF TokenType;
```

The individual delimiters, as specified in the `leftDelims` and `rightDelims` parameters, are paired by position. The first (in storage order) opening delimiter in `leftDelims` is paired with the first closing delimiter in `rightDelims`.

Comment delimiters may be 1 or 2 tokens each and there may be two sets of opening and closing pairs. They are passed to `IntlTokenize` in a `commentType` array:

```
TYPE CommentType = ARRAY[0..3] OF TokenType;
```

If only one token is needed for a delimiter, the second token must be specified to be `delimPad`. If only one delimiter of an opening-closing pair is needed, then both of the tokens allocated for the other symbol must be `delimPad`. The first token of a two-token sequence is at the higher position in the `leftComment` or `rightComment` array. For example, if the two opening (in this case, left) delimiters were "`(*`" and "`{`", they would be specified as follows:

```
leftComment[0] := tokenAsterisk;    (*asterisk*)
leftComment[1] := tokenLeftParen;   (*left parenthesis*)
leftComment[2] := delimPad ;        (*nothing*)
leftComment[3] := tokenLeftCurly;   (*curly brace*)
```

When `IntlTokenize` encounters an escape character within a quoted literal, it places the portion of the literal before the escape character into a single token (of type `tokenLiteral`), places the escape character into another token (`tokenEscape`), places the character following the escape character into another token (whatever token type it corresponds to), and places the portion of the literal following the escape sequence into another token (`tokenLiteral`). Outside of a quoted literal, the escape character has no special significance.

`IntlTokenize` considers the character specified in the `decimalCode` parameter to be a decimal character only when it is flanked by numeric or alternate numeric characters, or when it follows them.

SPECIAL CONSIDERATIONS

`IntlTokenize` may move memory; your application should not call this function at interrupt time.

Because each call to `IntlTokenize` must be for a single script run, there can be no change of script within a comment or quoted literal.

Comments and quoted literals must be complete within a single call to `IntlTokenize` in order to avoid syntax errors.

`IntlTokenize` always uses the tokens resource whose handle you pass it in the token block record. Therefore, it is not directly affected by the state of the font force flag or the international resources selection flag. However, if you use the `GetIntlResource` function to get a handle to the tokens resource to pass to `IntlTokenize`, remember that `GetIntlResource` is affected by the state of the international resources selection flag. See "Determining Script Codes From Font Information" beginning on page 6-21.

RESULT CODES

`tokenOK`	0	Valid token
`tokenOverflow`	1	Number of tokens exceeded maximum specified in `tokenList` field of token block record
`stringOverflow`	2	Size of string list larger than maximum specified in `stringList` field of token block record
`badDelim`	3	Invalid delimiter
`badEnding`	4	(currently unused)
`crash`	5	Unknown error

SEE ALSO

See the appendix "International Resources" in this book for a description of the tokens (`'itl4'`) resource.

Transliteration

Transliteration is the conversion of text from one form or subscript to another within a single script system. In the Roman script system, transliteration means case conversion. In 2-byte script systems, it is the automatic conversion of characters from one subscript to another. One common use for transliteration is as an initial stage of text conversion for an input method.

TransliterateText

The `TransliterateText` function converts characters from one subscript to the closest possible approximation in a different subscript within the same 2-byte script system. `TransliterateText` also performs uppercasing and lowercasing, with consideration for regional variants, in the Roman script system and on Roman text within 2-byte script systems.

```
FUNCTION TransliterateText (srcHandle: Handle;
                            dstHandle: Handle;
                            target: Integer; srcMask: LongInt;
                            script: ScriptCode): OSErr;
```

`srcHandle` A handle to the source text to be transliterated.

`dstHandle` A handle to a buffer that, upon completion of the call, contains the transliterated text.

`target` A value that specifies what kind of text the source text is to be transliterated into. The low byte of the target is the format to convert to. The high byte contains modifiers, whose meanings depend on the script code.

`srcMask` A bit array that specifies which parts of the source text are to be transliterated. A bit is set for each script system or subscript that should be converted.

`script` A value that specifies the script system of the text to be transliterated. Constants for all defined script codes are listed on page 6-52. To specify the font script, pass `smCurrentScript` in this parameter.

DESCRIPTION

The types of conversions `TransliterateText` performs are described in the section "Transliteration" beginning on page 6-43.

The `TransliterateText` function converts all of the text that you pass it in the `srcHandle` parameter. It determines the length of the source text (in bytes) from the handle size.

Before calling `TransliterateText`, allocate a handle (of any size) to pass in the `dstHandle` parameter. The length of the transliterated text may be different (as when converting between 1-byte and 2-byte characters), and `TransliterateText` sets the size of the destination handle as required. It is your responsibility to dispose of the destination handle when you no longer need it.

The `srcMask` parameter is the source mask; it specifies which subscript(s) represented in the source text should be converted to the target format. In all script systems, the `srcMask` parameter may have the following values: `smMaskAscii`, `smMaskNative`, and `smMaskAll`, as described on page 6-46. In 2-byte script systems, additional values are recognized, as described on page 6-46.

The low-order byte of the `target` parameter is the target format; it determines what form the the text should be transliterated to. In all script systems, there are two currently supported values for target format: `smTransAscii` and `smTransNative`, as described on page 6-46. In 2-byte script systems, additional values are recognized, as described on page 6-47.

The high-order byte of the `target` parameter is the target modifier; it provides additional formatting instructions. In all script systems, there are two values for target modifer: `smTransLower` and `smTransUpper`, as described on page 6-47.

Note

Because the low-ASCII character set (character codes $20–$7F) is present in all script systems, you could theoretically use the `TransliterateText` function to convert characters from one script system into another completely different script system. You could transliterate from a native subscript into ASCII under one script system, and then transliterate from that ASCII into a native subscript under a different script system. Such a procedure is not recommended, however, because of the imperfect nature of phonetic translation. Furthermore, many script systems do not support transliteration from native subscripts to ASCII. ◆

SPECIAL CONSIDERATIONS

`TransliterateText` may move memory; your application should not call this function at interrupt time.

If you pass `smCurrentScript` in the `script` parameter, the conversion performed by `TransliterateText` can be affected by the state of the font force flag. It is unaffected by the international resources selection flag.

Transliteration of a block of text does not work across script-run boundaries. Because the `TransliterateText` function requires transliteration tables that are in a script system's international resources, you need to call it anew for each script run in your text.

Currently, the Roman version of `TransliterateText` checks the source mask only to ensure that at least one of the bits corresponding to the `smMaskAscii` and `smMaskNative` constants is set.

The Arabic and Hebrew versions of `TransliterateText` perform case conversion only. They allow the target values `smTransAscii` and `smTransNative` only; otherwise, they behave like the Roman version.

The `TransliterateText` tables for 1-byte script systems reside in the script's string-manipulation (`'itl2'`) resource, so they can reflect region-specific or language-specific differences in uppercase conventions. If the string-manipulation resource does not include these tables, `TransliterateText` exits without doing anything.

The `TransliterateText` tables for 2-byte script systems reside in the script's transliteration (`'trsl'`) resource. If the `'trsl'` resource does not include these tables, `TransliterateText` exits without doing anything.

The Japanese, Traditional Chinese, and Simplified Chinese versions of `TransliterateText` have two modes of operation:

- If either `smMaskAscii` or `smMaskNative` is specified in the source mask, and if the target is `smTransAscii`, and if either of the target modifiers is specified, `TransliterateText` performs the specified case conversion on both 1-byte and 2-byte Roman letters.
- Otherwise, `TransliterateText` performs conversions according to the target format values defined on page 6-47. Any combination of source masks and target format is permitted.

RESULT CODES

In addition to Memory Manager errors, `TransliterateText` can return the following results:

`noErr`	0	No error
	–1	Illegal source or target, or `'itl2'` could not be loaded

Replacing a Script System's Default Routines

The four Script Manager routines described in this section allow you to access or replace the text-manipulation and text-display routines in WorldScript I, the system extension for 1-byte complex script systems. The function `GetScriptUtilityAddress` and the procedure `SetScriptUtilityAddress` work with the script utilities routines. The function `GetScriptQDPatchAddress` and the procedure `SetScriptQDPatchAddress` work with patches of the QuickDraw routines `StdText`, `StdTxMeas`, and `MeasureText`, and the Font Manager routine `FontMetrics`.

For more information on how to use these calls, see the appendix "Built-in Script Support" in this book.

For `GetScriptUtilityAddress` and `SetScriptUtilityAddress`, these are the valid values for the `selector` parameter:

Script utility	Selector value
`GetScriptVariable`	$000C
`SetScriptVariable`	$000E
`CharacterByteType`	$0010
`CharacterType`	$0012
`TransliterateText`	$0018
`FindWordBreaks`	$001A
`HiliteText`	$001C
`FillParseTable`	$0022
`FindScriptRun`	$0026
`VisibleLength`	$0028
`PixelToChar`	$002E
`CharToPixel`	$0030
`DrawJustified`	$0032
`Measurejustified`	$0034
`PortionLine`	$0036

For `GetScriptQDPatchAddress` and `SetScriptQDPatchAddress`, these are the valid values for the `trapNum` parameter:

QuickDraw patch	`trapNum` value
`_StdText`	$A882
`_StdTxMeas`	$A8ED
`_MeasureText`	$A837
`_FontMetrics`	$A835

GetScriptUtilityAddress

The `GetScriptUtilityAddress` function returns a pointer to the specified 1-byte script utility—or the original Roman utility—for the given script system.

```
FUNCTION GetScriptUtilityAddress (selector: Integer;
                                  before: Boolean;
                                  script: ScriptCode): Ptr;
```

`selector` A value that specifies the name of the utility routine whose address is needed.

`before` A Boolean that specifies which of two routines is needed. If `TRUE`, the address returned is that of the WorldScript I implementation of the utility. If `FALSE`, the address returned is that of the original routine (usually the built-in Roman version).

`script` The numeric code that specifies the script system whose dispatch table contains the pointers to the utility routines. Constants for all defined script codes are listed on page 6-52.

DESCRIPTION

The `GetScriptUtilityAddress` function examines the specified script's dispatch table and returns a pointer to the desired routine.

Because each element in the dispatch table consists of a pair of addresses, one for the WorldScript I implementation of the utility, and another for the original (Roman) version of the utility, you can get the address of either routine. Either routine can then be replaced, using the `GetScriptUtilityAddress` call.

This function can return `NIL` for the pointer if, for example, either the WorldScript I implementation or the original Roman routine is not used by the script system.

Valid values for the `selector` parameter are listed on page 6-101.

If the specified script system is not enabled, `GetScriptUtilityAddress` returns a `NIL` pointer.

SEE ALSO

WorldScript I is described in the appendix "Built-in Script Support" in this book.

SetScriptUtilityAddress

The `SetScriptUtilityAddress` procedure replaces the specified 1-byte script utility—or the original Roman utility—for the given script.

```
PROCEDURE SetScriptUtilityAddress (selector: Integer;
                                   before: Boolean;
                                   routineAddr: Ptr;
                                   script: ScriptCode);
```

`selector` A value that specifies the name of the utility routine to be replaced.

`before` A Boolean that specifies which of two routines is to be replaced. If `TRUE`, the WorldScript I implementation of the utility is replaced. If `FALSE`, the original routine (usually the built-in Roman version) is replaced.

`routineAddr` A pointer to the routine that is to replace the script utility.

`script` The numeric code that specifies the script system whose dispatch table contains the pointers to the utility routines. Constants for all defined script codes are listed on page 6-52.

DESCRIPTION

The `SetScriptUtilityAddress` procedure replaces the pointer to the desired routine in the specified script's dispatch table.

Several of the WorldScript I utilities call the original Roman routine after they execute. Each element in the dispatch table consists of a pair of addresses: one for the WorldScript I implementation of the utility, and another for the original (Roman) version of the utility. With `SetScriptUtilityAddress` you can replace either routine. Thus you can insert your patch code either before (or in place of) the WorldScript I version of the utility, or before (or in place of) the original Roman routine.

IMPORTANT

When you patch a script system's script utility, you alter that script's behavior for as long as it remains enabled. Therefore, be sure to restore the pointer to its original state whenever your application quits or is switched out by the Process Manager. ▲

Valid values for the `selector` parameter are listed on page 6-101.

SEE ALSO

WorldScript I is described in the appendix "Built-in Script Support" in this book.

GetScriptQDPatchAddress

The `GetScriptQDPatchAddress` function returns a pointer to the specified WorldScript I QuickDraw patch—or the built-in QuickDraw call—for the given script system.

```
FUNCTION GetScriptQDPatchAddress (trapNum: Integer;
                                  before: Boolean;
                                  forPrinting: Boolean;
                                  script: ScriptCode): Ptr;
```

`trapNum` A value that specifies the name of the QuickDraw routine whose address is needed.

`before` A Boolean that specifies which of two routines is needed. If `TRUE`, the address returned is that of the WorldScript I patch to the QuickDraw routine. If `FALSE`, the address returned is that of the original routine (usually the built-in QuickDraw routine).

`forPrinting`
A Boolean that specifies whether the desired routine is for printing. If `TRUE`, the address returned is that of a QuickDraw patch that is specifically for printing; if `FALSE`, the address returned is that of a QuickDraw patch that is not specifically for printing.

`script` The numeric code that specifies the script system whose dispatch table contains the pointers to the QuickDraw routines. Constants for all defined script codes are listed on page 6-52.

DESCRIPTION

The `GetScriptQDPatchAddress` function examines the specified script's dispatch table and returns a pointer to the desired routine.

Because each element in the dispatch table consists of a pair of addresses, one for the WorldScript I patch to the QuickDraw routine, and another for the original QuickDraw version of the routine, you can get the address of either routine. Either routine can then be replaced, using the `SetScriptQDPatchAddress` call.

Some printers perform their own text layout on text that is passed to them. Therefore, each QuickDraw patch has two entry points: one for screen display and printing, and one for printing only. By specifying either `TRUE` or `FALSE` in the `forPrinting` parameter, the pointer you obtain is to either the "for printing only" or the "not for printing only" entry point. For example, some script systems might use the "for printing only" entry point to perform extra-fine justification of text on a PostScript printer.

Valid values for the `trapNum` parameter are listed on page 6-101.

If the specified script system is not enabled, `GetScriptQDPatchAddress` returns a `NIL` pointer.

SEE ALSO

WorldScript I is described in the appendix "Built-in Script Support" in this book.

In order to handle contextual formatting appropriately for each script system, printer drivers should call the Script Manager's print action routine, described in *Inside Macintosh: Devices.*

SetScriptQDPatchAddress

The `SetScriptQDPatchAddress` procedure replaces the WorldScript I specified QuickDraw patch—or the built-in QuickDraw call—for the given script.

```
PROCEDURE SetScriptQDPatchAddress (trapNum: Integer;
                                   before: Boolean;
                                   forPrinting: Boolean;
                                   routineAddr: Ptr;
                                   script: ScriptCode);
```

`trapNum` A value that specifies the name of the QuickDraw routine that is to be replaced.

`before` A Boolean that specifies which of two routines is to be replaced. If `TRUE`, the WorldScript I patch of the QuickDraw routine is replaced. If `FALSE`, the original routine (usually the built-in QuickDraw routine) is replaced.

`forPrinting`
A Boolean that specifies whether the replacement routine is for printing. If `TRUE`, the new QuickDraw patch is specifically for printing; if `FALSE`, the new QuickDraw patch is not specifically for printing.

`routineAddr`
A pointer to the routine that is to replace the existing QuickDraw routine.

`script` The numeric code that specifies the script system whose dispatch table contains the pointers to the QuickDraw routines. Constants for all defined script codes are listed on page 6-52.

DESCRIPTION

The `SetScriptQDPatchAddress` procedure replaces the pointer to the desired routine in the specified script's dispatch table.

All of the WorldScript I patches call the original QuickDraw routine after they execute. Each element in the dispatch table consists of a pair of addresses: one for the WorldScript I patch, and another for the original (built-in QuickDraw) version of the routine. With `SetQDPatchAddress` you can replace either routine. Thus you can insert your patch code either before (or in place of) the WorldScript I QuickDraw patch, or before (or in place of) the original QuickDraw routine.

Some printers perform their own text layout on text that is passed to them. Therefore, each QuickDraw patch has two entry points: one for screen display and one for printing only. By specifying either `TRUE` or `FALSE` in the `forPrinting` parameter, you specify whether you are passing the "for printing only" or the "not for printing only" entry point. For example, some script systems might use the "for printing only" entry point to perform extra-fine justification of text on a PostScript printer.

IMPORTANT

When you patch a script system's QuickDraw call, you alter that script's behavior for as long as it remains enabled. Therefore, be sure to restore the pointer to its original state whenever your application quits or is switched out by the Process Manager. ▲

Valid values for the `trapNum` parameter are listed on page 6-101.

SEE ALSO

WorldScript I is described in the appendix "Built-in Script Support" in this book.

In order to handle contextual formatting appropriately for each script system, printer drivers should call the Script Manager's print action routine, described in *Inside Macintosh: Devices.*

Summary of the Script Manager

Pascal Summary

Constants

```
{Script system constants}

{Implicit script codes}
smSystemScript = -1;       {designates system script.}
smCurrentScript = -2;      {designates font script.}
smAllScripts = -3;         {designates any script}
{Explicit script codes}
smRoman = 0;               {Roman}
smJapanese = 1;            {Japanese}
smTradChinese = 2;         {Traditional Chinese}
smKorean = 3;              {Korean}
smArabic = 4;              {Arabic}
smHebrew = 5;              {Hebrew}
smGreek = 6;               {Greek}
smCyrillic = 7;            {Cyrillic}
smRSymbol = 8;             {Right-left symbol}
smDevanagari = 9;          {Devanagari}
smGurmukhi = 10;           {Gurmukhi}
smGujarati = 11;           {Gujarati}
smOriya = 12;              {Oriya}
smBengali = 13;            {Bengali}
smTamil = 14;              {Tamil}
smTelugu = 15;             {Telugu}
smKannada = 16;            {Kannada/Kanarese}
smMalayalam = 17;          {Malayalam}
smSinhalese = 18;          {Sinhalese}
smBurmese = 19;            {Burmese}
smKhmer = 20;              {Khmer/Cambodian}
smThai = 21;               {Thai}
smLaotian = 22;            {Laotian}
```

```
smGeorgian = 23;          {Georgian}
smArmenian = 24;          {Armenian}
smSimpChinese = 25;       {Simplified Chinese}
smTibetan = 26;           {Tibetan}
smMongolian = 27;         {Mongolian}
smGeez = 28;              {Geez/Ethiopic}
smEthiopic = 28;          {Synonym for smGeez}
smEastEurRoman = 29;      {Synonym for smSlavic}
smVietnamese = 30;        {Vietnamese}
smExtArabic = 31;         {extended Arabic}
smUninterp = 32;          {uninterpreted symbols, e.g. palette symbols}

{Language Codes}
langEnglish = 0;          { smRoman script }
langFrench = 1;           { smRoman script }
langGerman = 2;           { smRoman script }
langItalian = 3;          { smRoman script }
langDutch = 4;            { smRoman script }
langSwedish = 5;          { smRoman script }
langSpanish = 6;          { smRoman script }
langDanish = 7;           { smRoman script }
langPortuguese = 8;       { smRoman script }
langNorwegian = 9;        { smRoman script }
langHebrew = 10;          { smHebrew script }
langJapanese = 11;        { smJapanese script }
langArabic = 12;          { smArabic script }
langFinnish = 13;         { smRoman script }
langGreek = 14;           { smGreek script }
langIcelandic = 15;       { extended Roman script }
langMaltese = 16;         { extended Roman script }
langTurkish = 17;         { extended Roman script }
langCroatian = 18;        { Serbo-Croatian in extended Roman script }
langTradChinese = 19;     { Chinese in traditional characters }
langUrdu = 20;            { smArabic script }
langHindi = 21;           { smDevanagari script }
langThai = 22;            { smThai script }
langKorean = 23;          { smKorean script }
langLithuanian = 24;      { smEastEurRoman script }
langPolish = 25;          { smEastEurRoman script }
langHungarian = 26;       { smEastEurRoman script }
langEstonian = 27;        { smEastEurRoman script }
langLettish = 28;         { smEastEurRoman script }
```

```
langLatvian = 28;               { Synonym for langLettish }
langSaamisk = 29;               { extended Roman script }
langLappish = 29;               { Synonym for langSaamisk }
langFaeroese = 30;              { smRoman script }
langFarsi = 31;                 { smArabic script }
langPersian = 31;               { Synonym for langFarsi }
langRussian = 32;               { smCyrillic script }
langSimpChinese = 33;           { Chinese in simplified characters }
langFlemish = 34;               { smRoman script }
langIrish = 35;                 { smRoman script }
langAlbanian = 36;              { smRoman script }
langRomanian = 37;              { smEastEurRoman script }
langCzech = 38;                 { smEastEurRoman script }
langSlovak = 39;                { smEastEurRoman script }
langSlovenian = 40;             { smEastEurRoman script }
langYiddish = 41;               { smHebrew script }
langSerbian = 42;               { Serbo-Croatian in smCyrillic script }
langMacedonian = 43;            { smCyrillic script }
langBulgarian = 44;             { smCyrillic script }
langUkrainian = 45;             { smCyrillic script }
langByelorussian = 46;          { smCyrillic script }
langUzbek = 47;                 { smCyrillic script }
langKazakh = 48;                { smCyrillic script }
langAzerbaijani = 49;           { Azerbaijani in smCyrillic script }
langAzerbaijanAr = 50;          { Azerbaijani in smArabic script (Iran) }
langArmenian = 51;              { smArmenian script }
langGeorgian = 52;              { smGeorgian script }
langMoldovan = 53;              { smCyrillic script }
langMoldavian = 53;             { Synonym for langMoldovan }
langKirghiz = 54;               { smCyrillic script }
langTajiki = 55;                { smCyrillic script }
langTurkmen = 56;               { smCyrillic script }
langMongolian = 57;             { Mongolian in smMongolian script }
langMongolianCyr = 58;          { Mongolian in smCyrillic script }
langPashto = 59;                { smArabic script }
langKurdish = 60;               { smArabic script }
langKashmiri = 61;              { smArabic script }
langSindhi = 62;                { smExtArabic script }
langTibetan = 63;               { smTibetan script }
langNepali = 64;                { smDevanagari script }
langSanskrit = 65;              { smDevanagari script }
langMarathi = 66;               { smDevanagari script }
```

```
langBengali = 67;               { smBengali script }
langAssamese = 68;              { smBengali script }
langGujarati = 69;              { smGujarati script }
langPunjabi = 70;               { smGurmukhi script }
langOriya = 71;                 { smOriya script }
langMalayalam = 72;             { smMalayalam script }
langKannada = 73;               { smKannada script }
langTamil = 74;                 { smTamil script }
langTelugu = 75;                { smTelugu script }
langSinhalese = 76;             { smSinhalese script }
langBurmese = 77;               { smBurmese script }
langKhmer = 78;                 { smKhmer script }
langLao = 79;                   { smLaotian script }
langVietnamese = 80;            { smVietnamese script }
langIndonesian = 81;            { smRoman script }
langTagalog = 82;               { smRoman script }
langMalayRoman = 83;            { Malay in smRoman script }
langMalayArabic = 84;           { Malay in smArabic script }
langAmharic = 85;               { smEthiopic script }
langTigrinya = 86;              { smEthiopic script }
langGalla = 87;                 { smEthiopic script }
langOromo = 87;                 { synonym for langGalla }
langSomali = 88;                { smRoman script }
langSwahili = 89;               { smRoman script }
langRuanda = 90;                { smRoman script }
langRundi = 91;                 { smRoman script }
langChewa = 92;                 { smRoman script }
langMalagasy = 93;              { smRoman script }
langEsperanto = 94;             { extended Roman script }
langWelsh = 128;                { smRoman script }
langBasque = 129;               { smRoman script }
langCatalan = 130;              { smRoman script }
langLatin = 131;                { smRoman script }
langQuechua = 132;              { smRoman script }
langGuarani = 133;              { smRoman script }
langAymara = 134;               { smRoman script }
langTatar = 135;                { smCyrillic script }
langUighur = 136;               { smArabic script }
langDzongkha = 137;             { (lang of Bhutan) smTibetan script }
langJavaneseRom = 138;          { Javanese in smRoman script }
langSundaneseRom = 139;         { Sundanese in smRoman script }
```

```
{ Region codes }
verUS = 0;
verFrance = 1;
verBritain = 2;
verGermany = 3;
verItaly = 4;
verNetherlands = 5;
verFrBelgiumLux = 6;          {French for Belgium & Luxembourg}
verSweden = 7;
verSpain = 8;
verDenmark = 9;
verPortugal = 10;
verFrCanada = 11;
verNorway = 12;
verIsrael = 13;
verJapan = 14;
verAustralia = 15;
verArabia = 16;
verArabic = 16;               {synonym for verArabia}
verFinland = 17;
verFrSwiss = 18;              {Swiss French}
verGrSwiss = 19;              {Swiss German}
verGreece = 20;
verIceland = 21;
verMalta = 22;
verCyprus = 23;
verTurkey = 24;
verYugoCroatian = 25;         {Croatian system}
verIndiaHindi = 33;           {Hindi system for India}
verPakistan = 34;
verLithuania = 41;
verPoland = 42;
verHungary = 43;
verEstonia = 44;
verLatvia = 45;
verLapland = 46;
verFaeroeIsl = 47;
verIran = 48;
verRussia = 49;
verIreland = 50;           {English-language version for Ireland}
```

```
verKorea = 51;
verChina = 52;
verTaiwan = 53;
verThailand = 54;
minCountry = verUS;
maxCountry = verThailand;

{Calendar codes}
calGregorian = 0;
calArabicCivil = 1;
calArabicLunar = 2;
calJapanese = 3;
calJewish = 4;
calCoptic = 5;
calPersian = 6;

{Numeral codes}
intWestern = 0;
intArabic = 1;
intRoman = 2;
intJapanese = 3;
intEuropean = 4;
intOutputMask = $8000;

{ CharacterByteType byte types }
smSingleByte = 0;
smFirstByte = -1;
smLastByte = 1;
smMiddleByte = 2;

{CharacterType field masks}
smcTypeMask = $000F;
smcReserved = $00F0;
smcClassMask = $0F00;
smcOrientationMask = $1000;   {2-byte script glyph orientation}
smcRightMask = $2000;
smcUpperMask = $4000;
smcDoubleMask = $8000;

{Basic CharacterType character types}
smCharPunct = $0000;
smCharAscii = $0001;
smCharEuro = $0007;
smCharExtAscii = $0007;       {more correct synonym for smCharEuro}
```

```
{Additional CharacterType character types for script systems}
smCharKatakana = $0002;          {Japanese Katakana}
smCharHiragana = $0003;          {Japanese Hiragana}
smCharIdeographic = $0004;       {Hanzi, Kanji, Hanja}
smCharTwoByteGreek = $0005;      {2-byte Greek in Far East systems}
smCharTwoByteRussian = $0006;    {2-byte Cyrillic in Far East systems}
smCharBidirect = $0008;          {Arabic/Hebrew}
smCharContextualLR = $0009;      {contextual left-right: Thai, Indic scripts}
smCharNonContextualLR = $000A;   {noncontextual left-right: Cyrillic, Greek}
smCharHangul = $000C;            {Korean Hangul}
smCharJamo = $000D;              {Korean Jamo}
smCharBopomofo = $000E;          {Chinese Bopomofo (Zhuyinfuhao)}

{CharacterType classes for punctuation (smCharPunct)}
smPunctNormal = $0000;
smPunctNumber = $0100;
smPunctSymbol = $0200;
smPunctBlank = $0300;

{Additional CharacterType classes for punctuation in two-byte systems}
smPunctRepeat = $0400;           {repeat marker}
smPunctGraphic = $0500;          {line graphics}

{CharacterType Katakana and Hiragana classes for 2-byte systems}
smKanaSmall = $0100;             {small Kana character}
smKanaHardOK = $0200;            {can have dakuten}
smKanaSoftOK = $0300;            {can have dakuten or han-dakuten}

{CharacterType ideographic classes for 2-byte systems}
smIdeographicLevel1 = $0000;     {level 1 char}
smIdeographicLevel2 = $0100;     {level 2 char}
smIdeographicUser = $0200;       {user char}

{CharacterType Jamo classes for Korean systems}
smJamoJaeum = $0000;             {simple consonant char}
smJamoBogJaeum = $0100;          {complex consonant char}
smJamoMoeum = $0200;             {simple vowel char}
smJamoBogMoeum = $0300;          {complex vowel char}

{CharacterType glyph orientation for 2-byte systems}
smCharHorizontal = $0000;        {horizontal character form, or for both}
smCharVertical = $1000;          {vertical character form}
```

```
{CharacterType directions}
smCharLeft = $0000;
smCharRight = $2000;

{CharacterType case modifers}
smCharLower = $0000;
smCharUpper = $4000;

{CharacterType character size modifiers (1 or multiple bytes)}
smChar1byte = $0000;
smChar2byte = $8000;

{TransliterateText target types for Roman}
smTransAscii = 0;             {convert to ASCII}
smTransNative = 1;            {convert to font script}
smTransCase = $FE;            {convert case for all text}
smTransSystem = $FF;          {convert to system script}

{TransliterateText target types for 2-byte scripts}
smTransAscii1 = 2;            {1-byte Roman}
smTransAscii2 = 3;            {2-byte Roman}
smTransKana1 = 4;             {1-byte Japanese Katakana}
smTransKana2 = 5;             {2-byte Japanese Katakana}
smTransGana2 = 7;             {2-byte Japanese Hiragana (no 1-byte Hiragana)}
smTransHangul2 = 8;           {2-byte Korean Hangul}
smTransJamo2 = 9;             {2-byte Korean Jamo}
smTransBopomofo2 = 10;        {2-byte Chinese Bopomofo (Zhuyinfuhao)}

{TransliterateText target modifiers}
smTransLower = $4000;         {target becomes lowercase}
smTransUpper = $8000;         {target becomes uppercase}

{TransliterateText resource format numbers}
smTransRuleBaseFormat = 1;    {rule-based trsl resource format }
smTransHangulFormat = 2;      {table-based Hangul trsl resource format}

{TransliterateText property flags}
smTransPreDoubleByting = 1;   {convert all text to 2-byte }
                              { before transliteration}
smTransPreLowerCasing = 2;    {convert all text to lowercase }
                              { before transliteration}

{TransliterateText source mask - general}
smMaskAll = $FFFFFFFF;        {convert all text}
```

```
{TransliterateText source masks}
smMaskAscii = $00000001;         {2^smTransAscii}
smMaskNative = $00000002;        {2^smTransNative}

{TransliterateText source masks for 2-byte scripts}
smMaskAscii1 = $00000004;        {2^smTransAscii1}
smMaskAscii2 = $00000008;        {2^smTransAscii2}
smMaskKana1 = $00000010;         {2^smTransKana1}
smMaskKana2 = $00000020;         {2^smTransKana2}
smMaskGana2 = $00000080;         {2^smTransGana2}
smMaskHangul2 = $00000100;       {2^smTransHangul2}
smMaskJamo2 = $00000200;         {2^smTransJamo2}
smMaskBopomofo2 = $00000400;     {2^smTransBopomofo2}

{Result values from GetScriptManagerVariable, SetScriptManagerVariable, }
{ GetScriptVariable, and SetScriptVariable}
smNotInstalled = 0;              {routine not available in specified script}
smBadVerb = -1;                  {bad selector passed to a routine}
smBadScript = -2;                {bad script code passed to a routine}

{Values for script redraw flag}
smRedrawChar = 0;                {redraw character only}
smRedrawWord = 1;                {redraw entire word (2-byte systems)}
smRedrawLine = -1;               {redraw entire line (bidirectional systems)}

{GetScriptManagerVariable and SetScriptManagerVariable selectors}
smVersion = 0;                   {Script Manager version number}
smMunged = 2;                    {Globals change count}
smEnabled = 4;                   {Count of enabled scripts, incl Roman}
smBidirect = 6;                  {At least one bidirectional script}
smFontForce = 8;                 {Force font flag}
smIntlForce = 10;                {Intl resources selection flag}
smForced = 12;                   {Script was forced to system script}
smDefault = 14;                  {Script was defaulted to Roman script}
smPrint = 16;                    {Printer action routine}
smSysScript = 18;                {System script}
smLastScript = 20;               {Last keyboard script}
smKeyScript = 22;                {Keyboard script}
smSysRef = 24;                   {System folder refNum}

smKeyCache = 26;                 {obsolete}
smKeySwap = 28;                  {Swapping table handle}
smGenFlags = 30;                 {General flags long}
smOverride = 32;                 {Script override flags}
```

```
smCharPortion = 34;              {Ch vs SpExtra proportion}
smDoubleByte = 36;             {Flag for double-byte script installed}
smKCHRCache = 38;              {Returns pointer to KCHR cache}
smRegionCode = 40;             {Returns current region code (verXxx)}
smKeyDisableState = 42;        {Returns current keyboard disable state}

{ GetScriptVariable and SetScriptVariable selectors.}
smScriptVersion = 0;           {Script software version}
smScriptMunged = 2;            {Script entry changed count}
smScriptEnabled = 4;           {Script enabled flag}
smScriptRight = 6;             {Right to left flag}
smScriptJust = 8;              {Justification flag}
smScriptRedraw = 10;           {Word redraw flag}
smScriptSysFond = 12;          {Preferred system font}
smScriptAppFond = 14;          {Preferred Application font}
smScriptBundle = 16;           {Beginning of itlb verbs}
smScriptNumber = 16;           {Script itl0 id}
smScriptDate = 18;             {Script itl1 id}
smScriptSort = 20;             {Script itl2 id}
smScriptFlags = 22;            {flags word}
smScriptToken = 24;            {Script itl4 id}
smScriptEncoding = 26;         {id of optional itl5, if present}
smScriptLang = 28;             {Current language for script}
smScriptNumDate = 30;          {Script Number/Date formats.}
smScriptKeys = 32;             {Script KCHR id}
smScriptIcon = 34;             {ID # of SICN or kcs#/kcs4/kcs8 family}
smScriptPrint = 36;            {Script printer action routine}
smScriptTrap = 38;             {Trap entry pointer}
smScriptCreator = 40;          {Script file creator}
smScriptFile = 42;             {Script file name}
smScriptName = 44;             {Script name}
smScriptMonoFondSize = 78;     {default monospace FOND (hi) & size (lo)}
smScriptPrefFondSize = 80;     {preferred FOND (hi) & size (lo)}
smScriptSmallFondSize = 82;    {default small FOND (hi) & size (lo)}
smScriptSysFondSize = 84;      {default system FOND (hi) & size (lo)}
smScriptAppFondSize = 86;      {default app FOND (hi) & size (lo)}
smScriptHelpFondSize = 88;     {default Help Mgr FOND (hi) & size (lo)}
smScriptValidStyles = 90;      {mask of valid styles for script}
smScriptAliasStyle = 92;       {style (set) to use for aliases}

{ Negative selectors for KeyScript }
smKeyNextScript = -1;          { Switch to next available script }
smKeySysScript = -2;           { Switch to the system script }
smKeySwapScript = -3;          { Switch to previously-used script }
```

```
smKeyNextKybd = -4;     { Switch to next keyboard in current keyscript }
smKeySwapKybd = -5;     { Switch to previous keyboard in current keyscript }
smKeyDisableKybds = -6; { Disable keyboards not in system or Roman script }
smKeyEnableKybds = -7;  { Re-enable keyboards for all enabled scripts }
smKeyToggleInline = -8; { Toggle inline input for current keyscript }
smKeyToggleDirection = -9;    {Toggle default line direction (TESysJust)}
smKeyNextInputMethod = -10;   {Switch to next input method in current script}
smKeySwapInputMethod = -11;   {Switch to prev. input method in curr. script}
smKeyDisableKybdSwitch = -12; {Disable switching from current keyboard}
smKeySetDirLeftRight = -15;   {Set default line dir. left-right,align left}
smKeySetDirRightLeft = -16;   {Set default line dir. right-left,align right}
smKeyRoman = -17;       { Set keyscript to Roman. Does nothing if Roman-only}

{ Bits in the smScriptFlags word
(bits above 8 are non-static) }
smsfIntellCP = 0;          {Script has intelligent cut & paste}
smsfSingByte = 1;          {Script has only single bytes}
smsfNatCase = 2;           {Native chars have upper & lower case}
smsfContext = 3;           {Script is contextual}
smsfNoForceFont = 4;       {Script will not force characters}
smsfB0Digits = 5;          {Script has alternate digits at B0-B9}

smsfAutoInit = 6;          {Auto initialize the script}
smsfUnivExt = 7;           {Script is handled by WorldScript I}
smsfSynchUnstyledTE = 8;   {Synchronize keyboard and chartype in unstyled TE}
smsfForms = 13;            {Uses contextual forms for letters}
smsfLigatures = 14;        {Uses contextual ligatures}
smsfReverse = 15;          {Reverses native text, right-left}

{ Bits in the smGenFlags long.}
{First (high-order) byte is set from itlc flags byte. }
smfShowIcon = 31;          {Show icon even if only one script}
smfDualCaret = 30;         {Use dual caret for mixed direction text}
smfNameTagEnab = 29;       {Reserved for internal use}

{ Script Manager font equates. }
smFondStart = $4000;       {start from 16K}
smFondEnd = $C000;         {past end of range at 48K}

{ Miscellaneous font equates. }
smUprHalfCharSet = $80;    {first char code in top half of std char set}

{ Character Set Extensions }
diaeresisUprY = $D9;
fraction = $DA;
```

```
intlCurrency = $DB;
leftSingGuillemet = $DC;
rightSingGuillemet = $DD;
fiLigature = $DE;
flLigature = $DF;
dblDagger = $E0;
centeredDot = $E1;
baseSingQuote = $E2;
baseDblQuote = $E3;
perThousand = $E4;
circumflexUprA = $E5;
circumflexUprE = $E6;
acuteUprA = $E7;
diaeresisUprE = $E8;
graveUprE = $E9;
acuteUprI = $EA;
circumflexUprI = $EB;
diaeresisUprI = $EC;

graveUprI = $ED;
acuteUprO = $EE;
circumflexUprO = $EF;
appleLogo = $F0;
graveUprO = $F1;
acuteUprU = $F2;
circumflexUprU = $F3;
graveUprU = $F4;
dotlessLwrI = $F5;
circumflex = $F6;
tilde = $F7;
macron = $F8;
breveMark = $F9;
overDot = $FA;
ringMark = $FB;
cedilla = $FC;
doubleAcute = $FD;
ogonek = $FE;
hachek = $FF;

{ TokenType values }
tokenIntl = 4;              {the itl resource number of the tokenizer}
tokenEmpty = -1;            {used internally as an empty flag}
tokenUnknown = 0;           {chars that do not match a defined token type}
tokenWhite = 1;             {whitespace}
```

```
tokenLeftLit = 2;          {literal begin}
tokenRightLit = 3;         {literal end}
tokenAlpha = 4;            {alphabetic}
tokenNumeric = 5;          {numeric}
tokenNewLine = 6;          {new line}
tokenLeftComment = 7;      {open comment}
tokenRightComment = 8;     {close comment}
tokenLiteral = 9;          {literal}
tokenEscape = 10;          {character escape (e.g. '\' in "\n", "\t")}
tokenAltNum = 11;          {alternate number (e.g. $B0-B9 in Arabic,Hebrew)}
tokenRealNum = 12;         {real number}
tokenAltReal = 13;         {alternate real number}
tokenReserve1 = 14;        {reserved}
tokenReserve2 = 15;        {reserved}
tokenLeftParen = 16;       {open parenthesis}
tokenRightParen = 17;      {close parenthesis}

tokenLeftBracket = 18;     {open square bracket}
tokenRightBracket = 19;    {close square bracket}
tokenLeftCurly = 20;       {open curly bracket}
tokenRightCurly = 21;      {close curly bracket}
tokenLeftEnclose = 22;     {open guillemet}
tokenRightEnclose = 23;    {close guillemet}
tokenPlus = 24;
tokenMinus = 25;
tokenAsterisk = 26;        {times/multiply}
tokenDivide = 27;
tokenPlusMinus = 28;       {plus or minus symbol}
tokenSlash = 29;
tokenBackSlash = 30;
tokenLess = 31;            {less than symbol}
tokenGreat = 32;           {greater than symbol}
tokenEqual = 33;
tokenLessEqual2 = 34;      {less than or equal, 2 characters (e.g. <=)}
tokenLessEqual1 = 35;      {less than or equal, 1 character}
tokenGreatEqual2 = 36;     {greater than or equal, 2 characters (e.g. >=)}
tokenGreatEqual1 = 37;     {greater than or equal, 1 character}
token2Equal = 38;          {double equal (e.g. ==)}

tokenColonEqual = 39;      {colon equal}
tokenNotEqual = 40;        {not equal, 1 character}
tokenLessGreat = 41;       {less/greater, Pascal not equal (e.g. <>)}
tokenExclamEqual = 42;     {exclamation equal, C not equal (e.g. !=)}
tokenExclam = 43;          {exclamation point}
```

```
tokenTilde = 44;           {centered tilde}
tokenComma = 45;
tokenPeriod = 46;
tokenLeft2Quote = 47;      {open double quote}
tokenRight2Quote = 48;     {close double quote}
tokenLeft1Quote = 49;      {open single quote}
tokenRight1Quote = 50;     {close single quote}
token2Quote = 51;          {double quote}
token1Quote = 52;          {single quote}
tokenSemicolon = 53;
tokenPercent = 54;
tokenCaret = 55;
tokenUnderline = 56;
tokenAmpersand = 57;
tokenAtSign = 58;
tokenBar = 59;             {vertical bar}
tokenQuestion = 60;
tokenPi = 61;              {lower-case pi}
tokenRoot = 62;            {square root symbol}
tokenSigma = 63;           {capital sigma}
tokenIntegral = 64;        {integral sign}
tokenMicro = 65;
tokenCapPi = 66;           {capital pi}
tokenInfinity = 67;
tokenColon = 68;
tokenHash = 69;            {e.g. #}
tokenDollar = 70;
tokenNoBreakSpace = 71;    {non-breaking space}
tokenFraction = 72;
tokenIntlCurrency = 73;
tokenLeftSingGuillemet = 74;
tokenRightSingGuillemet = 75;
tokenPerThousand = 76;
tokenEllipsis = 77;
tokenCenterDot = 78;
tokenNil = 127;
delimPad = -2;

{ Table selectors for GetIntlResourceTable }
smWordSelectTable = 0;     { get word break table from 'itl2' }
smWordWrapTable = 1;       { get line break table from 'itl2' }
smNumberPartsTable = 2;    { get number parts table from 'itl4' }
smUnTokenTable = 3;        { get unToken table from 'itl4' }
smWhiteSpaceList = 4;      { get whitespace table from 'itl4' }
```

Data Types

```
TYPE  TokenResults =
      (tokenOK,tokenOverflow,stringOverflow,badDelim,
      badEnding,crash);

      CharByteTable = PACKED ARRAY[0..255] OF SignedByte;

      TokenType = Integer;

      DelimType = ARRAY[0..1] OF TokenType;

      CommentType = ARRAY[0..3] OF TokenType;

      TokenRec =
      RECORD
         theToken:         TokenType;
         position:         Ptr;        {pointer into original source}
         length:           LongInt;    {length of text in original source}
         stringPosition:   StringPtr;  {Pascal/C string copy of identifier}
      END;
      TokenRecPtr = ^TokenRec;

      TokenBlock =
      RECORD
         source:           Ptr;     {pointer to stream of characters}
         sourceLength:     LongInt; {length of source stream}
         tokenList:        Ptr;     {pointer to array of tokens}
         tokenLength:      LongInt; {maximum length of TokenList}
         tokenCount:       LongInt; {number tokens generated by tokenizer}
         stringList:       Ptr;     {pointer to stream of identifiers}
         stringLength:     LongInt; {length of string list}
         stringCount:      LongInt; {number of bytes currently used}
         doString:         Boolean; {make strings & put into StringList}
         doAppend:         Boolean; {append to TokenList rather than replace}

         doAlphanumeric:   Boolean; {identifiers may include numeric}
         doNest:           Boolean; {do comments nest?}
         leftDelims:       DelimType;
         rightDelims:      DelimType;
         leftComment:      CommentType;
         rightComment:     CommentType;
         escapeCode:       TokenType;{escape symbol code}
         decimalCode:      TokenType;
```

```
      itlResource:      Handle;  {handle to current script itl4 resource}
      reserved:         ARRAY [0..7] OF LongInt;   {must be zero!}
   END;
   TokenBlockPtr = ^TokenBlock;
```

Routines

Checking and Setting the System Direction

```
FUNCTION GetSysDirection: Integer;
PROCEDURE SetSysDirection   (newDirection: Integer);
```

Checking and Setting Script Manager Variables

```
FUNCTION GetScriptManagerVariable
                            (selector: Integer): LongInt;
FUNCTION SetScriptManagerVariable
                            (selector: Integer; param: LongInt): OSErr;
```

Checking and Setting Script Variables

```
FUNCTION GetScriptVariable  (script: ScriptCode;
                             selector: Integer): LongInt;
FUNCTION SetScriptVariable  (script: ScriptCode; selector: Integer;
                             param: LongInt): OSErr;
```

Making Keyboard Settings

```
PROCEDURE KeyScript         (code: Integer);
```

Determining Script Codes From Font Information

```
FUNCTION FontScript: Integer;
FUNCTION FontToScript       (fontNumber: Integer): Integer;
FUNCTION IntlScript: Integer;
```

Analyzing Characters

```
FUNCTION CharacterByteType  (textBuf: Ptr;textOffset: Integer;
                             script: ScriptCode): Integer;
FUNCTION CharacterType      (textBuf: Ptr; textOffset: Integer;
                             script: ScriptCode): Integer;
FUNCTION FillParseTable     (VAR table: CharByteTable;
                             script: ScriptCode): Boolean;
```

Directly Accessing International Resources

```
PROCEDURE ClearIntlResourceCache;
FUNCTION GetIntlResource    (theID: Integer): Handle;
PROCEDURE GetIntlResourceTable
                            (script: ScriptCode;tableCode: Integer;VAR
                             itlHandle: Handle; VAR offset: LongInt;VAR
                             length: LongInt);
```

Tokenization

```
FUNCTION IntlTokenize       (tokenParam: TokenBlockPtr): TokenResults;
```

Transliteration

```
FUNCTION TransliterateText  (srcHandle: Handle; dstHandle: Handle;
                             target: Integer; srcMask: LongInt;
                             script: ScriptCode): OSErr;
```

Replacing a Script System's Default Routines

```
FUNCTION GetScriptUtilityAddress
                            (selector: Integer; before: Boolean;
                             script: ScriptCode): Ptr;
PROCEDURE SetScriptUtilityAddress
                            (selector: Integer; before: Boolean;
                             routineAddr: Ptr; script: ScriptCode);
FUNCTION GetScriptQDPatchAddress
                            (trapNum: Integer;
                             before: Boolean; forPrinting: Boolean;
                             script: ScriptCode): Ptr;
PROCEDURE SetScriptQDPatchAddress
                            (trapNum: Integer;before: Boolean;
                             forPrinting: Boolean; routineAddr: Ptr;
                             script: ScriptCode);
```

C Summary

Constants

Please see page 6-107 for a listing of constants defined in Pascal by the Script Manager. The constants as defined in C are identical to them.

Data Types

```
typedef unsigned char TokenResults;

typedef char CharByteTable[256];

typedef short TokenType;

typedef TokenType DelimType[2];

typedef TokenType CommentType[4];

struct TokenRec {
 TokenType theToken;
 Ptr position;               /*pointer into original source*/
 long length;                /*length of text in original source*/
 StringPtr stringPosition;   /*Pascal/C string copy of identifier*/
};
typedef struct TokenRec TokenRec;
typedef TokenRec *TokenRecPtr;

struct TokenBlock {
 Ptr source;                 /*pointer to stream of characters*/
 long sourceLength;          /*length of source stream*/
 Ptr tokenList;              /*pointer to array of tokens*/
 long tokenLength;           /*maximum length of TokenList*/
 long tokenCount;            /*number tokens generated by tokenizer*/
 Ptr stringList;             /*pointer to stream of identifiers*/
 long stringLength;          /*length of string list*/
 long stringCount;           /*number of bytes currently used*/
 Boolean doString;           /*make strings & put into StringList*/
 Boolean doAppend;           /*append to TokenList rather than replace*/
 Boolean doAlphanumeric;     /*identifiers may include numeric*/
 Boolean doNest;             /*do comments nest?*/
```

```
 TokenType leftDelims[2];
 TokenType rightDelims[2];
 TokenType leftComment[4];
 TokenType rightComment[4];
 TokenType escapeCode;        /*escape symbol code*/
 TokenType decimalCode;
 Handle itlResource;          /*handle to itl4 resource of current script*/
 long reserved[8];            /*must be zero!*/
};
typedef struct TokenBlock TokenBlock;
typedef TokenBlock *TokenBlockPtr;
```

Routines

Checking and Setting the System Direction

```
#define GetSysDirection()   (* (short*) 0x0BAC);
pascal void SetSysDirection (short newDirection);
```

Checking and Setting Script Manager Variables

```
pascal long GetScriptManagerVariable
                            (short selector);
pascal OSErr SetScriptManagerVariable
                            (short selector, long param);
```

Checking and Setting Script Variables

```
pascal long GetScriptVariable
                            (ScriptCode script, short selector);
pascal OSErr SetScriptVariable
                            (ScriptCode script, short selector, long param);
```

Making Keyboard Settings

```
pascal void KeyScript       (short code);
```

Determining Script Codes From Font Information

```
pascal short FontScript     (void);
pascal short FontToScript   (short fontNumber);
pascal short IntlScript     (void);
```

Analyzing Characters

```
pascal short CharacterByteType
                            (Ptr textBuf, short textOffset,
                             ScriptCode script);
pascal short CharacterType  (Ptr textBuf, short textOffset,
                             ScriptCode script);
pascal Boolean FillParseTable
                            (CharByteTable table, ScriptCode script);
```

Directly Accessing International Resources

```
pascal void ClearIntlResourceCache
                            (void);
pascal Handle GetIntlResource
                            (short theID);
pascal void GetIntlResourceTable
                            (ScriptCode script, short tableCode,
                             Handle *itlHandle, long *offset, long *length);
```

Tokenization

```
pascal TokenResults IntlTokenize
                            (TokenBlockPtr tokenParam);
```

Transliteration

```
pascal OSErr TransliterateText
                            (Handle srcHandle, Handle dstHandle,
                             short target, long srcMask, ScriptCode script);
```

Replacing a Script System's Default Routines

```
pascal Ptr GetScriptUtilityAddress
                            (short selector, Boolean before,
                             ScriptCode script);
pascal void SetScriptUtilityAddress
                            (short selector, Boolean before,
                             Ptr routineAddr, ScriptCode script);
pascal Ptr GetScriptQDPatchAddress
                            (short trapNum, Boolean before,
                             Boolean forPrinting, ScriptCode script);
pascal void SetScriptQDPatchAddress
                            (short trapNum, Boolean before,
                             Boolean forPrinting, Ptr routineAddr,
                             ScriptCode script);
```

Assembly-Language Summary

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
FontScript	_FontScript
IntlScript	_IntlScript
KeyScript	_KeyScript
FontToScript	_FontToScript
GetScriptManagerVariable	_GetScriptManagerVariable
SetScriptManagerVariable	_SetScriptManagerVariable
GetScriptVariable	_GetScriptVariable
SetScriptVariable	_SetScriptVariable
CharacterByteType	_CharacterByteType
CharacterType	_CharacterType
TransliterateText	_TransliterateText
FillParseTable	_FillParseTable
GetScriptUtilityAddress	_GetScriptUtilityAddress
SetScriptUtilityAddress	_SetScriptUtilityAddress
GetScriptQDPatchAddress	_GetScriptQDPatchAddress
SetScriptQDPatchAddress	_SetScriptQDPatchAddress
IntlTokenize	_IntlTokenize
GetIntlResource	_GetIntlResource
ClearIntlResourceCache	_ClearIntlResourceCache
GetIntlResourceTable	_GetIntlResourceTable

Global Variables

SysDirection	System direction; the primary line direction and alignment for text
BootDrive	The drive number of the startup volume