International Resources

This appendix describes the international resources, which constitute the major portion of each Macintosh script system. The international resources define how a script system implements its particular writing system and how it allows for language or regional variations within a writing system.

The Script Manager, the Text Utilities, QuickDraw, and the Font Manager all use the international resources directly to handle text in various script systems. TextEdit makes indirect use of information in the international resources through calls to the Script Manager and other managers.

A text application uses the international resources indirectly whenever it makes a call to a script-aware routine in QuickDraw, the Text Utilities, or the Script Manager. It can also access the international resources directly through Script Manager calls, in order to

- pass a resource handle or pointer as a parameter to a text-handling routine
- extract formatting information from a table within a resource
- modify the contents of a resource, to customize text handling

Your most common reason to access the international resources may be to get a handle or pointer to pass to a text-handling routine. For that task, you do not need the information in this appendix.

Read this appendix if your application needs information about the internal structure of one or more international resources. If you need a particular resource table to perform a specific operation, such as formatting currencies or dates, extracting number parts, or converting script-independent tokens to the text of a particular script system, this appendix shows you where to get the information you need.

Read this appendix also if your application requires a custom localized version of some text-handling feature. To provide that feature, you can modify one or more of the international resources and supply that modified version with your application or its documents. In this way, you can localize the formats of numbers, currency, time, dates, and measurement; you can localize string comparison and word selection; you can modify the conversion of strings to tokens; you can specify custom character-rendering behavior; and you can specify custom transliteration rules.

Read this appendix also if you are creating a new script system. A complete script system requires a full set of the appropriate international resources, certain keyboard resources (as described in the appendix "Keyboard Resources" in this book), and one or more fonts.

Before reading this appendix, read the chapter "Introduction to Text on the Macintosh" in this book. The parts of the Macintosh script management system that make use of the resources documented here are described in the chapters "QuickDraw Text," "Text Utilities," and especially "Script Manager," in this book. The Resource Manager, which manages all Macintosh resources, is described in *Inside Macintosh: More Macintosh Toolbox*.

This appendix describes the international resources in general, shows the relationship between resource ID and script code, shows how to gain access to international resources and use them, and then describes each resource in detail.

About the International Resources

This section introduces the international resources, describes how the set of international resources varies among script systems and among different localized versions of Macintosh system software, and gives a table showing the relationship between script code and resource ID number used by fonts and by specific types of international resources and keyboard resources.

What the International Resources Are

Each script system consists of a set of international resources and a set of keyboard resources. These resources, possibly in conjunction with the WorldScript I or WorldScript II extension—and with the use of the proper font—completely specify a script's behavior. Because script-specific behavior is segregated into resources that are customizable and replaceable, your software can potentially use the same routines to handle text in any language, even one that is not curently supported.

The international resources that define individual script systems can include all but two in the following table. Two of the international resources, the international configuration resource and the script-sorting resource, are unique to each Macintosh System file; they do not belong to any script system.

Table B-1 lists the international resources and their resource types, and gives a capsule description of their contents. More complete descriptions follow.

Table B-1 The international resources

Name	**Resource type**	**Partial contents**
International configuration	`'itlc'`	Configuration of the system
Script sorting	`'itlm'`	Sorting order among scripts
International bundle	`'itlb'`	IDs of all resources for a script system
Numeric format	`'itl0'`	Number, time, and short-date formats
Long-date format	`'itl1'`	Long date formats
String manipulation	`'itl2'`	Sorting order, word breaks
Tokens	`'itl4'`	Tables of tokens, number parts
Encoding/rendering	`'itl5'`	Character encoding or rendering
Transliteration	`'trsl'`	Tables for phonetic conversion

- International configuration resource. Sets up the basic configuration for the system, including the system script. Specifies the system script code and the region code that identifies the regional version of the system script; initializes the states of the system direction, the font force flag, the international resources selection flag, the international keyboard flag (used for the Macintosh Plus), and the Script Manager general flags. There is only one `'itlc'` resource for each localized version of system software.
- Script-sorting resource. Specifies the preferred sorting order for script codes, language codes, and region codes. Also specifies the default language for each script, the parent script for each language, and the parent language for each region. There is only one `'itlm'` resource for each localized version of system software.
- International bundle resource. Sets up the basic configuration for an individual script system. The international bundle resource specifies the resource IDs for the script's resources. It also initializes many script variables, such as the script flags, the default language code, and the numeral and calendar representation codes for the script. The international bundle resource also specifies font information, script initialization data, valid styles for the script, and the style to use for designating aliases. Each script system has one `'itlb'` resource.
- Numeric-format resource. Contains short date and time formats, and formats for currency and numbers and the preferred unit of measurement. It also contains the region code for this particular resource. A script system can have one or more `'itl0'` resources.
- Long-date-format resource. Specifies the long date format for a particular region, including the names of days and months. Each long-date-format resource contains the region code for this particular resource. A long-date-format resource can have an optional extension for additional month and day names as well as abbreviated month and day names. A script system can have one or more `'itl1'` resources.
- String-manipulation resource. Contains routines that control text-sorting behavior, and tables for character type, case conversion, and word breaks. A script system can have one or more `'itl2'` resources.
- Tokens resource. Contains tables and code for converting text to tokens. It also has tables for formatting numbers, for converting tokens to text, and for determining whitespace characters. A script system can have one or more `'itl4'` resources.
- Encoding/rendering resource. Contains either information related to character encoding, or information controlling text-rendering behavior, in a script-specific format. This is an optional resource; a script system can have zero or more `'itl5'` resources.
- Transliteration resource. Specifies how to convert characters from one subscript to another within a script system. This is an optional resource; a script system can have zero or more `'trsl'` resources.

International resources and localized system software
When Macintosh system software is localized for a non-U.S. market, it contains replacements for or modifications to some of the U.S. versions of the international resources. See the discussion of U.S. international resources and keyboard resources in the appendix “Built-in Script Support” for a list of resources that may be replaced during localization. ◆

Script Codes and Resource ID Ranges

Fonts, international resources, and keyboard resources that are related to a particular script system have resource ID numbers in a range specific to that script. The script management system uses this relationship between resource ID and script code to assign the proper resources for displaying and formatting text. For example, the Script Manager `FontScript`, `IntlScript`, and `FontToScript` functions all use a font family ID to determine the script code that they return. Many other Script Manager, Text Utilities, and QuickDraw routines that load and use international resources take an explicit or implicit script code parameter; they will load only resources with ID numbers in the proper range for the supplied script code.

This numbering convention applies to font family IDs, and to the ID numbers of the following international and keyboard resources: `'itl0'`, `'itl1'`, `'itl2'`, `'itl4'`, `'itl5'`, `'trsl'`, `'KCHR'`, `'itlk'`, `'kcs#'`, `'kcs4'`, and `'kcs8'`.

Resources with ID numbers below 16384 ($4000) belong to the Roman script system. Currently, the script management system uses the following formula to calculate the script code for resources with ID numbers of 16384 and over:

```
scriptCode = ((resourceID - 16384) DIV 512) + 1
```

The formula allots half of the range of nonnegative ID values to the Roman script system, and 512 ID numbers (for each resource type) to each other script system, as shown in Table B-2. Please note that this formula may change in the future; note also that future script systems may possibly use negative ID values.

Table B-2 Resource ID ranges for each script system

Script system	Script code	Resource ID range	
		Decimal	Hexadecimal
Roman	0	2–16383	$0000–$3FFF
Japanese	1	16384–16895	$4000–$41FF
Chinese (Traditional)	2	16896–17407	$4200–$43FF
Korean	3	17408–17919	$4400–$45FF

Table B-2 Resource ID ranges for each script system (continued)

Script system	**Script code**	**Resource ID range**	
		Decimal	Hexadecimal
Arabic	4	17920–18431	$4600–$47FF
Hebrew	5	18432–18943	$4800–$4FF9
Greek	6	18944–19455	$4A00–$4BFF
Russian	7	19456–19967	$4C00–$4DFF
Right-left symbols	8	19968–20479	$4E00–$4FFF
Devanagari	9	20480–20991	$5000–$51FF
Gurmukhi	10	20992–21503	$5200–$53FF
Gujarati	11	21504–22015	$5400–$55FF
Oriya	12	22016–22527	$5600–$57FF
Bengali	13	22528–23039	$5800–$5FF9
Tamil	14	23040–23551	$5A00–$5BFF
Telugu	15	23552–24063	$5C00–$5DFF
Kannada	16	24064–24575	$5E00–$5FFF
Malayalam	17	24576–25087	$6000–$61FF
Sinhalese	18	25088–25599	$6200–$63FF
Burmese	19	25600–26111	$6400–$65FF
Cambodian	20	26112–26623	$6600–$67FF
Thai	21	26624–27135	$6800–$6FF9
Laotian	22	27136–27647	$6A00–$6BFF
Georgian	23	27648–28159	$6C00–$6DFF
Armenian	24	28160–28671	$6E00–$6FFF
Chinese (Simplified)	25	28672–29183	$7000–$71FF
Tibetan	26	29184–29695	$7200–$73FF
Mongolian	27	29696–30207	$7400–$75FF
Ethiopian	28	30208–30719	$7600–$77FF
Non-Cyrillic Slavic	29	30720–31231	$7800–$79FF
Vietnamese	30	31232–31743	$7A00–$7BFF
Sindhi	31	31744–32255	$7C00–$7DFF
Uninterpreted symbols	32	32256–32767	$7E00–$7FFF

Starting with a script code, you can back-calculate resource ID ranges as follows:

- Scripts with script codes in the range 1–32 have a range of 512 resource ID numbers, beginning with a number calculated according to this formula:

  ```
  firstID = 16384 + 512 * (script code – 1)
  ```

- Scripts with script codes in the range 33–64 have a range of 512 resource ID numbers, beginning with a number calculated according to this formula:

  ```
  firstID = –32768 + 512 * (script code –33)
  ```

Note
Some script codes above 32 are not usable because they correspond to resource ID ranges that are reserved for other purposes. Script codes 33 through 40 are invalid; furthermore, script codes above 48 are currently unavailable and may become invalid. ◆

Constants for all defined script codes are listed in the chapter "Script Manager" in this book.

Using the International Resources

The Script Manager and the other managers that make up the Macintosh script management system use the information in the international resources to format dates and times, find word boundaries, transliterate text, and determine character type, among other tasks. Your application indirectly accesses that information when it makes script-aware calls that rely on the current script system. In addition, you can directly access an international resource in order to

- pass a resource handle or pointer as a parameter to a text-handling routine. Many text-handling calls may take an explicit handle to an international resource; you first load the resource with calls to the Script Manager, and then pass its handle as a parameter to the call.
- extract formatting information from a table within a resource. If you are formatting currencies, dates, or numbers (without calling the Text Utilities routines that do formatting for you) , or if you are converting script-independent tokens to the text of a particular script system, you can load the appropriate resource with calls to the Script Manager, and then examine its contents for the information you need.
- provide a modified version of a resource, to customize text handling. You can load the appropriate resource with calls to the Script Manager, change it, and then save the changed resource in such a manner that it is used in place of the original resource.

Keep these points in mind when using a script system's international resources:

- You can load the international resources `'itl0'`, `'itl1'`, `'itl2'`, or `'itl4'` directly with `GetResource` or other related Resource Manager routines, but it is not recommended. If you use a Script Manager call such as `GetIntlResource` instead, the Script Manager determines which particular instance of an international resource to load, given the current font script, the script system's default preferences, and the current state of the international resources selection flag.

- Remember that most of the script-specific international resources have ID numbers within a range unique to their script system. If you are providing resources that add to or replace a script system's default resources, make sure that your resources have resource IDs in the proper range.
- If the international resources selection flag is set to `TRUE`, the international resources used by several script-aware Text Utilities routines are those of the system script. However, you can force those routines to use the international resources of the font script instead by clearing the international resources selection flag to `FALSE`. You can set and clear the international resources selection flag by using the Script Manager `SetScriptManagerVariable` function. See the discussion on determining script codes in the chapter "Script Manager" in this book.
- You can use multiple formats for different languages or regions with the same script system by adding multiple versions of international resources, each having a different resource ID within the script's range. You store those international resources in your application's or document's resource fork, where they can override those in the System file.

For more information, see the discussions of direct access to international resources and replacing default international resources in the chapter "Script Manager" in this book.

Note
Several international resources have type definitions that give you direct access to their components from high-level languages. These definitions are documented in this appendix. For other international resources high-level types are not defined, and graphic figures show the structures instead. ◆

International Configuration Resource (Type 'itlc')

The international configuration resource (resource type `'itlc'`) contains script-related configuration information for the currently executing version of system software. Only one `'itlc'` resource is provided with each localized version of Macintosh system software. It is in the System file. Its resource ID is 0.

The Script Manager uses the international configuration resource at startup to configure the system and the system script. The resource includes fields that specify these attributes:

- script code for the system script
- region code for this localized version of system software and the system script
- initial values for the Script Manager general flags
- initial value for the system direction

Several Script Manager variables are initialized from this resource. Selectors for the Script Manager variables are listed in the chapter "Script Manager" in this book.

The ItlcRecord Data Type

The international configuration record (data type `ItlcRecord`) describes the contents of the international configuration resource.

```
TYPE ItlcRecord =
   RECORD
      itlcSystem:       Integer;       {system script}
      itlcReserved:     Integer;       {reserved}
      itlcFontForce:    SignedByte;    {initial font force flag}
      itlcIntlForce:    SignedByte;    {initial int'l res. flag}
      itlcOldKybd:      SignedByte;    {old keyboard}
      itlcFlags:        SignedByte;    {Script Mgr. general flags}
      itlcIconOffset:   Integer;       {reserved}
      itlcIconSide:     SignedByte;    {reserved}
      itlcIconRsvd:     SignedByte;    {reserved}
      itlcRegionCode:   Integer;       {preferred region code}
      itlcSysFlags:     Integer;       {flags for system globals}
      itlcReserved4:    ARRAY [0..31] OF SignedByte;  {reserved}
   END;
```

Field descriptions

`itlcSystem`	The script code defining the system script. The system script affects system default settings, such as the default font and the text that appears in dialog boxes and menu bars, and so forth. Script codes and their constants are listed in the chapter "Script Manager" in this book. At startup, this value is copied into the Script Manager variable accessed through the selector `smSysScript`.
`itlcReserved`	Reserved.
`itlcFontForce`	The initial setting for the font force flag. A value of `TRUE` ($FF) forces Roman fonts to be interpreted as belonging to the system script. The font force flag is described in the chapter "Script Manager" in this book. At startup, this value is copied into the Script Manager variable accessed through the selector `smFontForce`.
`itlcIntlForce`	The initial setting for the international resources selection flag. A value of `TRUE` ($FF) forces Text Utilities routines to use the international resources for the system script, rather than the font script. The international resources selection flag is described in the chapter "Script Manager" in this book. At startup, this value is copied into the Script Manager variable accessed through the selector `smIntlForce`.

`itlcOldKybd`	The initial setting for the international keyboard flag for use by the Macintosh Plus computer. In addition to the standard Macintosh Plus keyboard (keyboard type 11), two types of smaller keyboard without numeric keypad are available: a U.S. version and an international version. Both have a keyboard type of 3, and the user uses the Keyboard control panel to indicate which is being used; the user's selection is saved in this field. When `TRUE` ($FF), this flag indicates that the international keyboard is being used. When `FALSE`, this flag indicates that the U.S. keyboard is being used.
`itlcFlags`	The initial settings for the Script Manager general flags. At startup, this value is copied into the first (high-order) byte of the Script Manager variable accessed through the selector `smGenFlags`.
`itlcIconOffset`	(reserved).
`itlcIconSide`	(reserved).
`itlcIconRsvd`	(reserved).
`itlcRegionCode`	The region code for this version of system software. It specifies the region for which the system and system-script resources were localized. The constants that define region codes are also described in the chapter "Script Manager" in this book. At startup, this value is copied into the Script Manager variable accessed through the selector `smRegionCode`.
`itlcSysFlags`	Flags for setting system global variables. Currently only one bit is defined; it allows the configuration resource to set the system direction (left-to-right or right-to-left) at startup. It is bit 15, defined by the following constant:

```
CONST
   itlcSysDirection     = 15;
```

The system global `SysDirection` is initialized from this value. A value of 0 for bit 15 sets a system direction of left-to-right (`SysDirection` = 0) at startup, whereas a value of 1 for the bit sets a system direction of right-to-left (`SysDirection` = $FFFF). You can access `SysDirection` through the Script Manager routines `GetSysDirection` and `SetSysDirection`. System direction may be initially localized to a value appropriate for the system script, but the user can reset its value at any time if a bidirectional script system is present.

Note

The `itlcSysFlags` field was formerly the `itlcReserved3` field. ◆

`itlcReserved4`	An array of 32 bytes that is reserved for future use.

Script-Sorting Resource (Type 'itlm')

The script-sorting resource (resource type `'itlm'`) lists, in preferred sorting order, a set of script codes, language codes, and region codes. For each listed script system it defines the default language; for each listed language it defines the script system that language belongs to; and for each listed regional version it describes the language that region belongs to. The listing may be sparse; not all defined script, language, and region codes need appear in the resource. Only one script-sorting resource is provided with each localized version of Macintosh system software. It is in the System file. Its resource ID is 0.

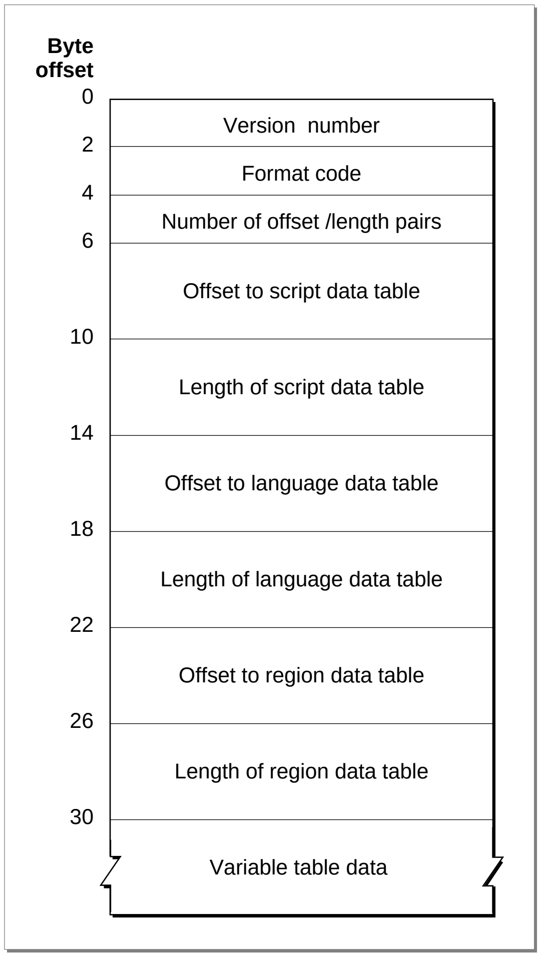
One purpose of the script-sorting resource is to aid the sorting of multilanguage lists. Each individual script system defines, in its string-manipulation (`'itl2'`) resource, how its own strings are sorted; the script-sorting resource defines how strings in two or more different scripts (or languages or regions) are ordered. For example, the string-manipulation resource for the Japanese script system defines the order in which Japanese strings appear in a sorted list. The script-sorting resource, on the other hand, defines whether Japanese strings appear before or after Roman strings in a sorted list.

IMPORTANT
Regardless of the sorting order presented in the script-sorting resource, text in the system script is always sorted to appear ahead of text in any other script system. ▲

Another purpose of the script-sorting resource is to provide a mapping among scripts, languages, and regions. From information in the resource you can determine all the languages of a listed script system, and all the regional variations of a listed language.

The script-sorting resource consists of a resource header followed by three tables. Figure B-1 shows the format of the resource header.

Figure B-1 Format of the script-sorting resource header

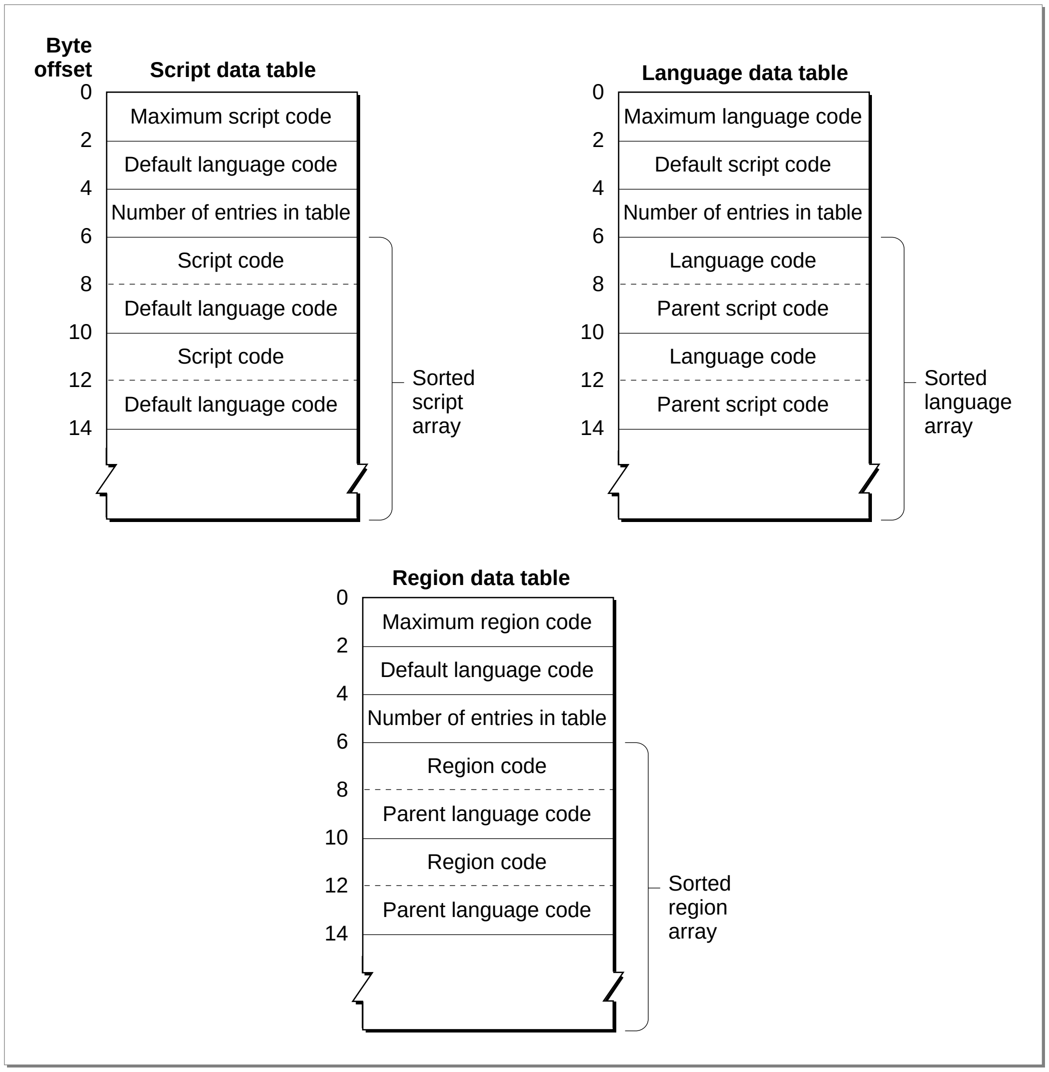

Byte offset
0
Version number
2
Format code
4
Number of offset /length pairs
6
Offset to script data table
10
Length of script data table
14
Offset to language data table
18
Length of language data table
22
Offset to region data table
26
Length of region data table
30
Variable table data

The resource header contains these elements:

- Version number. The version number of this resource.
- Format code. A number that identifies the format of this resource.
- Number of offset/length pairs. The number of data tables in the resource.
- Offsets to, and lengths of, the defined tables for this resource. Offsets are measured from the beginning of the resource.

Currently there are three defined tables in the script-sorting resource: the script data table, the language data table, and the region data table. The formats of the three tables are similar, as shown in Figure B-2.

Figure B-2 Script, language, and region data tables in the script-sorting resource



Each table consists of a header, followed by an array of paired integers. These are the fields in the script data table, language data table, and region data table, respectively:

- Maximum code. The maximum defined value for script, language, or region code listed in this table. Because entries in the table may be sparse (incomplete), this value is useful for defining the maximum size of table to construct to hold the information. For example, a maximum script code of `smUninterp` means the script data table might cover any subset of the scripts with codes 0 through 32, but the table does not contain any script codes above 32.
- Default code. The default language code for unlisted script codes, the default script code for unlisted language codes, or the default language code for unlisted region codes. This assures a defined sorting position for any script, language, or region code, whether or not it is listed in the resource.
- Number of entries in table. The number of script codes, language codes, or region codes in this table.
- Sorted array. A list of paired integers, in sorting order:
  - For the script data table, it is a script array: a list of script codes in their preferred sorting order, each paired with (followed by) its default language code.
  - For the language data table, it is a language array: a list of language codes in their preferred sorting order, each paired with (followed by) the code for its parent script.
  - For the region data table, it is a region array: a list of region codes in their preferred sorting order, each paired with (followed by) the code for its parent language.

Constants for all defined script codes, language codes, and region codes are listed in the chapter “Script Manager” in this book.

Table B-3 lists a sorting hierarchy of scripts, languages, and regions generated from a sample script-sorting resource. Not all scripts and languages are represented in this list because region codes do not currently exist for all language codes and script codes.

Table B-3 Sorted scripts, languages, and regions from a script-sorting resource

Script code	Language code	Region code
`smRoman`	`langEnglish`	`verUS`
		`verBritain`
		`verAustralia`
		`verIreland`
	`langFrench`	`verFrance`
		`verFrBelgiumLux`
		`verFrCanada`
		`verFrSwiss`

continued

Table B-3 Sorted scripts, languages, and regions from a script-sorting resource (continued)

Script code	Language code	Region code
	`langGerman`	`verGermany`
		`verGrSwiss`
	`langItalian`	`verItaly`
	`langDutch`	`verNetherlands`
	`langSwedish`	`verSweden`
	`langSpanish`	`verSpain`
	`langDanish`	`verDenmark`
	`langPortuguese`	`verPortugal`
	`langNorwegian`	`verNorway`
	`langFinnish`	`verFinland`
	`langIcelandic`	`verIceland`
	`langMaltese`	`verMalta`
	`langTurkish`	`verTurkey`
	`langLithuanian`	`verLithuania`
	`langEstonian`	`verEstonia`
	`langLettish`	`verLatvia`
	`langSaamisk`	`verLapland`
	`langFaeroese`	`verFaeroeIsl`
	`langCroatian`	`verYugoCroatian`
`smEastEurRoman`	`langPolish`	`verPoland`
	`langHungarian`	`verHungary`
`smGreek`	`langGreek`	`verGreece`
`smCyrillic`	`langRussia`	`verRussia`
`smArabic`	`langArabic`	`verArabic`
	`langUrdu`	`verPakistan`
	`langFarsi`	`verIran`
`smHebrew`	`langHebrew`	`verIsrael`
`smDevanagari`	`langHindi`	`verIndiaHindi`

Table B-3 Sorted scripts, languages, and regions from a script-sorting resource (continued)

Script code	**Language code**	**Region code**
`smThai`	`langThai`	`verThailand`
`smTradChinese`	`langTradChinese`	`verTaiwan`
`smSimpChinese`	`langSimpChinese`	`verChina`
`smJapanese`	`langJapanese`	`verJapan`
`smKorean`	`langKorea`	`verKorea`

International Bundle Resource (Type ' itlb')

The international bundle resource (resource type `'itlb'`) has two purposes. First, it is the **bundle resource** for a particular script system: by analogy with the Finder bundle resource type, it specifies the resource IDs for the other international resources and keyboard resources used by that script. (See the Finder Interface chapter of *Inside Macintosh: Macintosh Toolbox Essentials* for a description of Finder bundle resources.) Second, the `'itlb'` resource contains configuration information for the script.

Several script variables are initialized from this resource. Selectors for the script variables are listed in the chapter “Script Manager” in this book. If you need to change the initial values of those variables, you need to change the content of the international bundle resource itself. For example, to change the initial keyboard layout (script variable `smScriptKeys`) for a script system, you would change the value of the `itlbKeys` field of the international bundle resource. The user can makes this change from the Keyboard control panel; the user can make other changes to the `'itlb'` resource from other control panels, as described under user control of script systems in the chapter “Introduction to Text on the Macintosh” in this book.

Each script system has one and only one international bundle resource. The resource ID of the resource is that script system’s script code. Therefore, once you know the script code for a particular script system, you can find all of the script’s default international and keyboard resources by examining the international bundle resource whose ID equals that script code. For a list of defined script codes, see the chapter “Script Manager” in this book.

The original international bundle resource, defined by the `ItlbRecord` data type, was defined for system software versions earlier than 7.0. The extended `'itlb'` record, defined by the `ItlbExtRecord` data type, is defined for system software versions 7.0 and later. It includes the standard international bundle resource and adds extensions.

The ItlbRecord Data Type

The structure of the standard international bundle resource, defined by the `ItlbRecord` data type, is as follows:

```
TYPE ItlbRecord =
   RECORD
      itlbNumber:    Integer;       {'itl0' ID number}
      itlbDate:      Integer;       {'itl1' ID number}
      itlbSort:      Integer;       {'itl2' ID number}
      itlbFlags:     Integer;       {script flags}
      itlbToken:     Integer;       {'itl4' ID number}
      itlbEncoding:  Integer;       {'itl5' ID number (optional)}
      itlbLang:      Integer;       {current language for script}
      itlbNumRep:    SignedByte;    {current numeral code}
      itlbDateRep:   SignedByte;    {current calendar code}
      itlbKeys:      Integer;       {'KCHR' ID number}
      itlbIcon:      Integer;       {ID of keyboard icon family}
   END;
```

Field descriptions

`itlbNumber`	The resource ID of the numeric-format (`'itl0'`) resource to be used by this script. The Script Manager initializes the script variable accessed through the selector `smScriptNumber` from this field.
`itlbDate`	The resource ID of the long-date-format (`'itl1'`) resource to be used by this script. The Script Manager initializes the script variable accessed through the selector `smScriptDate` from this field.
`itlbSort`	The resource ID of the string-manipulation (`'itl2'`) resource to be used by this script system. The Script Manager initializes the script variable accessed through the selector `smScriptSort` from this field.
`itlbFlags`	The bit flags that describe the features of this script system. The Script Manager initializes the script flags variable, accessed through the selector `smScriptFlags`, from this field. For example, you can set the `smsfAutoInit` bit in the `itlbFlags` field to instruct the Script Manager to initialize the script system automatically. For definitions of the constants that specify the components of the script flags word, see the list of selectors for script variables in the chapter “Script Manager” in this book.

`itlbToken`	The resource ID of the tokens (`'itl4'`) resource to be used by this script. The Script Manager initializes the script variable accessed through the selector `smScriptToken` from this field.
`itlbEncoding`	The resource ID of the encoding/rendering (`'itl5'`) resource to be used by this script system. The Script Manager initializes the script variable accessed through the selector `smScriptEncoding` from this field. If there is no encoding/rendering resource for this script, this field is set to 0. This field was reserved in versions of system software prior to 7.0.
`itlbLang`	The language code specifying the default language for this script. The Script Manager initializes the script variable accessed through the selector `smScriptLang` from this field. See the chapter "Script Manager" in this book for a list of defined language codes.
`itlbNumRep`	The numeral code to be used by this script system. This byte specifies which types of numerals the script supports. The Script Manager initializes the high-order byte of the script variable accessed through the selector `smScriptNumDate` from this field. For definitions of the constants that specify numeral codes, see the list of selectors for script variables in the chapter "Script Manager" in this book.
`itlbDateRep`	The calendar code to be used by this script system. This byte specifies which types of calendars the script supports. The Script Manager initializes the low-order byte of the script variable accessed through the selector `smScriptNumDate` from this field. For definitions of the constants that specify calendar codes, see the list of selectors for script variables in the chapter "Script Manager" in this book.
`itlbKeys`	The resource ID of the preferred keyboard-layout (`'KCHR'`) resource to be used by this script system. The Script Manager initializes the script variable accessed through the selector `smScriptKeys` from this field.
`itlbIcon`	The resource ID of the keyboard icon family (resource types `'kcs#'`, `'kcs4'`, and `'kcs8'`) for the default keyboard layout to be used with this script. The Script Manager initializes the script variable accessed through the selector `smScriptIcon` from this field. (When loading a keyboard-layout resource, the Script Manager in fact ignores that variable and looks only for a keyboard icon suite whose ID matches that of the keyboard-layout resource being loaded.)

The ItlbExtRecord Data Type

The extended `'itlb'` record adds font and style information to the standard `'itlb'` record. Its structure, defined by the `ItlbExtRecord` data type, is as follows:

```
TYPE ItlbExtRecord =
   RECORD
      base:             ItlbRecord;    {standard ItlbRecord}
      itlbLocalSize:    LongInt;       {size of script variables}
      itlbMonoFond:     Integer;       {default monospaced font}
      itlbMonoSize:     Integer;       {default monospaced size}
      itlbPrefFond:     Integer;       {not used}
      itlbPrefSize:     Integer;       {not used}
      itlbSmallFond:    Integer;       {default small font}
      itlbSmallSize:    Integer;       {default small font size}
      itlbSysFond:      Integer;       {default system font}
      itlbSysSize:      Integer;       {default system font size}
      itlbAppFond:      Integer;       {default application font}
      itlbAppSize:      Integer;       {default appl. font size}
      itlbHelpFond:     Integer;       {default Help font}
      itlbHelpSize:     Integer;       {default Help font size}
      itlbValidStyles:  Style;         {valid styles for script}
      itlbAliasStyle:   Style;         {styles to mark aliases}
   END;
```

Field descriptions

`base`	The standard `'itlb'` record for this script.
`itlbLocalSize`	The size of the record of script variables for this script system. (A script system whose `smsfAutoInit` bit in its `itlbFlags` field is set needs to provide this information for the Script Manager.)
`iltbMonoFond`	The font family ID for the preferred font for monospaced text in this script system. The Script Manager initializes the high-order word of the script variable accessed through the selector `smScriptMonoFondSize` from this field.
`itlbMonoSize`	The default size for monospaced text in this script system. The Script Manager initializes the low-order word of the script variable accessed through the selector `smScriptMonoFondSize` from this field.
`itlbPrefFond`	This field is currently unused.
`itlbPrefSize`	This field is currently unused.

`itlbSmallFond`	The font family ID for the default font to display small text in this script system. The Script Manager initializes the high-order word of the script variable accessed through the selector `smScriptSmallFondSize` from this field.
`itlbSmallSize`	The default size for small text in this script system. The Script Manager initializes the low-order word of the script variable accessed through the selector `smScriptSmallFondSize` from this field.
`itlbSysFond`	The font family ID for the preferred system font in this script system. The Script Manager initializes the high-order word of the script variable accessed through the selector `smSysFondSize` from this field.
`itlbSysSize`	The default size for the system font in this script system. The Script Manager initializes the script variable accessed through the selector `smScriptSysFond`, and the low-order word of the script variable accessed through the selector `smSysFondSize`, from this field.
`itlbAppFond`	The font family ID for the preferred application font in this script system. The Script Manager initializes the script variable accessed through the selector `smScriptAppFond`, and the high-order word of the script variable accessed through the selector `smScriptAppFondSize`, from this field.
`itlbAppSize`	The default size for the application font in this script system. The Script Manager initializes the low-order word of the script variable accessed through the selector `smScriptAppFondSize` from this field.
`itlbHelpFond`	The font family ID for the preferred font for Balloon Help in this script system. The Script Manager initializes the high-order word of the script variable accessed through the selector `smScriptHelpFondSize` from this field.
`itlbHelpSize`	The default size for the Balloon Help font in this script system. The Script Manager initializes the low-order word of the script variable accessed through the selector `smScriptHelpFondSize` from this field.
`itlbValidStyles`	A style code that defines all of the valid styles for this script system. (In a style code, the bit corresponding to each QuickDraw style is set if that style is valid for the specified script. For example, the extend style is not valid in the Arabic script system.) The Script Manager initializes the script variable accessed through the selector `smScriptValidStyles` from this field.
`itlbAliasStyle`	A style code that defines the styles to use for displaying alias names in this script system. For example, in the Roman script system, alias names are displayed in italics. The Script Manager initializes the script variable accessed through the selector `smScriptAliasStyle` from this field.

Numeric-Format Resource (Type 'itl0')

The numeric-format resource (resource type `'itl0'`) contains general conventions for formatting numeric strings. It provides separators for decimals, thousands, and lists; it determines currency symbols and units of measurement; it specifies formats for currency, times, and short dates (the specification of dates in purely numeric representation—for example, in the U.S. Roman script system the short date for Tuesday, December 3, 1946, is 12/3/46). It also contains the region code for this particular instance of the `'itl0'` resource.

Each enabled script system has one or more numeric-format resources. The resource ID for each one is within the range of resource ID numbers for that script system. The default numeric-format resource for a script is specified in the `itlbNumber` field of the script's international bundle (`'itlb'`) resource.

The Text Utilities routines `TimeString`, `LongTimeString`, and `StringToTime` use information in the numeric-format resource to create time strings and to convert time strings to internal numeric representations. See the chapter "Text Utilities" in this book. The Operating System Utilities function `IsMetric` examines the numeric-format resource to determine the result it returns. See *Inside Macintosh: Operating System Utilities.*

Each numeric-format resource specifies the following:

- Number format. The characters to use as the decimal separator, thousands separator, and list separator.
- Currency format. The currency symbol and its position; whether or not to include leading unit zero or trailing decimal zero; how to show negative values.
- Short date format. The order of presentation of the day, month, and year elements; whether or not to include the century and a leading zero for month or days; the separator for the elements.
- Time format. Whether or not to present leading zeros for hours, minutes, and seconds; whether to use a 24-hour time cycle or a 12-hour A.M./P.M. cycle; how to specify a trailing string (such as a morning or an evening string if a 12-hour time cycle is being used).
- Unit of measure. Whether or not the metric system should be used.

Table B-4 lists constants that you can use in the numeric-format and long-date-format resources to specify separators for standard international formats. For example, in the U.S., `slashSymbol` is the separator for the short date 12/3/46, but in Germany `periodSymbol` is the separator for the short date 3.12.1946.

Table B-4 Constants for specifying numeric separators

Constant	Symbol
`periodSymbol`	.
`commaSymbol`	,
`semicolonSymbol`	;
`dollarsignSymbol`	$
`slashSymbol`	/
`colonSymbol`	:

IMPORTANT
When it specifies the order of elements, the numeric-format resource describes them in terms of *storage order,* not display order. Using the information in a numeric-format resource frees you from assuming a particular memory order for the components of numbers and short dates. However, the resource does not necessarily specify the left-to-right order for displaying the components. ▲

The Intl0Rec Data Type

You can access the numeric-format resource through the `Intl0Rec` data type.

```
TYPE  Intl0Rec =
   PACKED RECORD
      decimalPt:     Char;    {decimal point character}
      thousSep:      Char;    {thousands separator}
      listSep:       Char;    {list separator}
      currSym1:      Char;    {currency symbol}
      currSym2:      Char;
      currSym3:      Char;
      currFmt:       Byte;    {currency format flags}
      dateOrder:     Byte;    {order of short date elements}
      shrtDateFmt:   Byte;    {short date format flags}
      dateSep:       Char;    {date separator}
      timeCycle:     Byte;    {time cycle:0-23, 0-11, or 12-11}
      timeFmt:       Byte;    {time format flags}
      mornStr:       PACKED ARRAY[1..4] OF Char;   {trailing }
                              { string for first 12-hour cycle}
      eveStr:        PACKED ARRAY[1..4] OF Char;   {trailing }
                              { string for last 12-hour cycle}
      timeSep:       Char;    {time separator}
      time1Suff:     Char;    {trailing string for morning }
      time2Suff:     Char;    { part of 24-hour cycle}
      time3Suff:     Char;
      time4Suff:     Char;
      time5Suff:     Char;    {trailing string for afternoon }
      time6Suff:     Char;    { part of 24-hour cycle}
      time7Suff:     Char;
      time8Suff:     Char;
      metricSys:     Byte;    {255 if metric, 0 if not}
      intl0Vers:     Integer; {version information}
   END;
   Intl0Ptr       = ^Intl0Rec;
   Intl0Hndl      = ^Intl0Ptr;
```

Note
A `NULL` character (ASCII code 0) in a field of type `Char` means that no such character exists. The currency symbol and the trailing string for the 24-hour cycle are separated into individual `Char` fields because of Pascal packing conventions. All strings include any required spaces. ◆

Field descriptions

`decimalPt`	Part of the number format definition. The1-byte character that appears before the decimal representation of a fraction with a denominator of 10. In the United States, this format is a period. In several European countries, it is a comma.
`thousSep`	Part of the number format definition. The 1-byte character that separates every three digits to the left of the decimal point. In the United States, this format is a comma. In several European countries, it is a period.
`listSep`	Part of the number format definition. The 1-byte character that separates numbers, as when a list of numbers is entered by the user; it must be different from the decimal point character. If it's the same as the thousands separator, the user must not include the latter in entered numbers. In the United States, this format is a semicolon. In the United Kingdom, it is a comma.
`currSym1`	Part of the currency format definition. The initial byte used to indicate currency. One character is sufficient for the United States ($) and United Kingdom (£).
`currSym2`	Part of the currency format definition. The second byte used to indicate currency. Two characters are required for France (Fr).
`currSym3`	Part of the currency format definition. The third byte used to indicate currency. Three characters are required for Italy (Li.) and Germany (DM.).

`currFmt` Part of the currency format definition. The four least significant bits are unused. The four most significant bits are Boolean values. Bit 7 determines whether there is a leading integer zero; for example, a 1 in this field specifies a format like 0.23, whereas a 0 specifies .23. Bit 6 determines whether there are trailing decimal zeros; for example, a 1 in this field specifies a format like 325.00, whereas a 0 specifies 325. Bit 5 determines whether to use a minus sign or parentheses to denote a negative currency amount; for example, a 1 in this field specifies a format like –0.45, whereas a 0 specifies (0.45). Bit 4 determines whether the currency symbol trails or leads; for example, a value of 1 in this field specifies a format like the $3.00 used in the United States, whereas a value of 0 specifies the 3 DM. used in Germany.

You can use the following predefined constants as masks to set or test the bits in the `currFmt` field:

Constant	Value	Explanation
`currSymLead`	16	Currency symbol leads
`currNegSym`	32	Use minus sign for negative
`currTrailingZ`	64	Use trailing decimal zeros
`currLeadingZ`	128	Use leading integer zero

Note

You can also apply the currency format’s leading-zero and trailing-zero indicators to the number format if desired. ◆

`dateOrder` Part of the short date format definition. Defines the order of the elements (month, day, and year) of the short date format. The order varies from region to region—for example, 12/29/72 is a common order in the United States, whereas 29.12.72 is common in Europe.

You can indicate the order of the day, month, and year with the following constants:

Constant	Value	Explanation
`mdy`	0	Month-day-year
`dmy`	1	Day-month-year
`ymd`	2	Year-month-day
`myd`	3	Month-year-day
`dym`	4	Day-year-month
`ydm`	5	Year-day-month

`shrtDateFmt`	Part of the short date format definition. The five least significant bits are unused. The three most significant bit fields are Boolean values that determine whether to show the century, and whether to show leading zeros in month and day numbers. For example, if the first bit is set to 1 it specifies a date format like 10/21/1917, and set to 0 specifies the format 10/21/17. The second bit set to 1 specifies a format like 05/23/84, and set to 0 specifies the format 5/23/84. The third bit set to 1 specifies a format like 12/03/46, and set to 0 specifies the format 12/3/46.

To set or test the bits in the `shrtDateFmt` field, you can use the following predefined constants as masks:

Constant	Value	Explanation
`dayLdingZ`	32	Show leading zero for day
`mntLdingZ`	64	Show leading zero for month
`century`	128	Show century

`dateSep`	Part of the short date format definition. The 1-byte character that separates the different parts of the date. For example, in the United States this character is a slash (12/3/46), in Italy it is a hyphen (3-12-46), and in Japan it is a decimal point (46.12.3).
`timeCycle`	Part of the time format definition. Indicates the time cycle—that is, whether to use 12 or 24 hours as the basis of time, and whether to consider midnight and noon to be 12:00 or 0:00. You can use the following predefined constants to specify the time cycle:

Constant	Value	Explanation
`timeCycle24`	0	Use 24-hour format (midnight = 0:00)
`timeCycleZero`	1	Use A.M./P.M. format (midnight and noon = 0:00)
`timeCycle12`	255	Use A.M./P.M. format (midnight and noon = 12:00)

`timeFmt`	Part of the time format definition. Indicates whether to show leading zeros in time representation. Bit 5 determines whether there are leading zeros in seconds; for example, a value of 1 in this field specifies a format like 11:15:05, whereas a 0 specifies the format 11:15:5. Bit 6 determines whether there are leading zeros in minutes; for example, a value of 1 in this field specifies a format like 10:05, whereas a 0 specifies the format 10:5. Bit 7 determines whether there are leading zeros in hours; for example, a value of 1 in this field specifies a format like 09:15, whereas a 0 specifies the format 9:15.

You can use the following predefined constants as masks for setting or testing bits in the time format field:

Constant	Value	Explanation
`secLeadingZ`	32	Use leading zero for seconds
`minLeadingZ`	64	Use leading zero for minutes
`hrLeadingZ`	128	Use leading zero for hours

`mornStr` Part of the time format definition. A string of up to 4 bytes to follow the time to indicate morning (for example, " AM"). Typically, the string includes a leading space.

`eveStr` Part of the time format definition. A string of up to 4 bytes to follow the time to indicate evening (for example, " PM"). Typically, the string includes a leading space.

`timeSep` Part of the time format definition. The 1-byte character that is the time separator (for example, the colon).

`time1Suff`, `time2Suff`, `time3Suff`, `time4Suff`
Part of the time format definition. A trailing string of up to 4 bytes for the morning part of the 24-hour cycle. For example, the German string "uhr" can be stored here.

`time5Suff`, `time6Suff`, `time7Suff`, `time8Suff`
Part of the time format definition. A trailing string of up to 4 bytes for the evening part of the 24-hour cycle. Typically, this string duplicates the string contained in `time1Suff` through `time4Suff`. For example, the German string "uhr" can be stored here.

`metricSys` The unit-of-measure definition. Indicates whether to use the metric system. If 255, the metric system is used; if 0, metric is not used.

`intl0Vers` Region code and version number. The code number of the region that this resource applies to is in the high-order byte, and the version number of this numeric-format resource is in the low-order byte.

Long-Date-Format Resource (Type 'itl1')

The long-date-format resource (resource type `'itl1'`) contains the long date format for a particular region, including the names of days and months, the exact order of presentation of the elements, and the specification of whether or not to suppress any element. (For example, in U.S. format, the long date for 12/3/46 without the name of the day suppressed is Tuesday, December 3, 1946.) The long-date-format resource also has an optional extension for additional month and day names as well as abbreviated month and day names and separators. The extension also specifies the calendar code (for example, Arabic lunar) to use for long dates.

Each enabled script system has one or more long-date-format resources. The resource ID for each one is within the range of resource ID numbers for that script system. The default long-date-format resource for a script system is specified in the `itlbDate` field of the script's international bundle (`'itlb'`) resource.

The Text Utilities routines `DateString`, `LongDateString`, and `StringToDate` use information in the long-date-format resource to create date strings and to convert date strings to internal numeric representations. See the chapter "Text Utilities" in this book.

The basic (unextended) long-date-format resource specifies

■ The long date format. Month and day names, order of elements, and which elements to suppress.

■ Separator strings. What characters (commonly punctuation) appear between elements of the date.

■ Region code. The number that identifies the regional version of the script system that this long-date-format resource applies to.

IMPORTANT
When it specifies the order of elements, the long-date-format resource describes them in terms of *storage order,* not display order. Using the information in a long-date-format resource frees you from assuming a particular memory order for the components of long dates. However, the resource does not necessarily specify the left-to-right order for displaying the components. ▲

The Intl1Rec Data Type

You can access the contents of the long-date-format resource through the `Intl1Rec` data type.

```
TYPE Intl1Rec =
PACKED RECORD
   days:          ARRAY[1..7] OF Str15;   {day names}
   months:        ARRAY[1..12] OF Str15;  {month names}
   suppressDay:   Byte;                   {elements to suppress}
   lngDateFmt:    Byte;                   {order of elements}
   dayLeading0:   Byte;                   {leading 0 in day no.?}
   abbrLen:       Byte;                   {abbreviation length}
   st0:           PACKED ARRAY[1..4] OF Char;   {separator string}
   st1:           PACKED ARRAY[1..4] OF Char;   {separator string}
   st2:           PACKED ARRAY[1..4] OF Char;   {separator string}
   st3:           PACKED ARRAY[1..4] OF Char;   {separator string}
   st4:           PACKED ARRAY[1..4] OF Char;   {separator string}
   intl1Vers:     Integer;                {version & region}
   localRtn:      ARRAY[0..0] OF Integer; {flag for extended itl1}
END;
TYPE Intl1Ptr = ^Intl1Rec;
TYPE Intl1Hndl = ^Intl1Ptr;
```

Field descriptions

`days` An array of 7 day names (ordered for days corresponding to Sunday through Saturday). Each day name may consist of a maximum of 15 characters.

`months` An array of 12 month names (ordered for months corresponding to January through December). Each month name may consist of a maximum of 15 characters.

`suppressDay` A byte that lets you omit any element in the long date. To include the day name in the long date, you set the field to 0. To suppress the day name, set the field to 255 ($FF).

If the value does not equal 0 or $FF, this field is treated as bit flags. You can use the following predefined constants as masks to set the appropriate bits in the `suppressDay` byte.

Constant	Value	Explanation
`supDay`	1	Suppress day of month
`supWeek`	2	Suppress day name
`supMonth`	4	Suppress month
`supYear`	8	Suppress year

Note that a value of 2 is same as a value of $FF in this field.

`lngDateFmt`	The byte that indicates the order of long date elements. If the byte value of the field is neither 0 (which specifies an order of day/month/year) nor $FF (which specifies an order of month/day/year), then its value is divided into 4 fields of 2 bits each. The least significant bit field (bits 0 and 1) corresponds to the first element in the long date format, whereas the most significant bit field (bits 6 and 7) specifies the last (fourth) element in the format. You can use the following predefined constants to set each bit field to the appropriate value.

Constant	Value	Explanation
`longDay`	0	Day of the month
`longWeek`	1	Day of the week
`longMonth`	2	Month
`longYear`	3	Year

Note that these constants represent values for the 2-bit field, and are neither masks nor bit numbers. For example, suppose you wanted long dates to appear in this order: day of the week, day of the month, month, and year. You would set the value of `longDateFmt` like this:

```
longDateFmt :=
   longWeek*1     {sets bits 0 and 1}
   + LongDay*4    {sets bits 2 and 3}
   + longMonth*16 {sets bits 4 and 5}
   + longYear*64; {sets bits 6 and 7}
```

`dayLeading0`	If 255 ($FF), specifies a leading zero in a day number. If 0, no leading zero is included in the day number.
`abbrLen`	The number of characters to which month and day names should be abbreviated when abbreviation is desired.
`st0`	String that precedes (in memory) the first element in a long date. See Table B-5 and Figure B-3.
`st1`	String that separates the first and second elements in a long date. See Table B-5 and Figure B-3. This string is suppressed if the first element in the long date is suppressed.
`st2`	String that separates the second and third elements in a long date. See Table B-5 and Figure B-3. This string is suppressed if the second element in the long date is suppressed.
`st3`	String that separates the third and fourth elements in a long date. See Table B-5 and Figure B-3. This string is suppressed if the third element in the long date is suppressed.

`st4`	String that follows the fourth element in a long date. See Table B-5 and Figure B-3. This string is suppressed if the fourth element in the long date is suppressed.

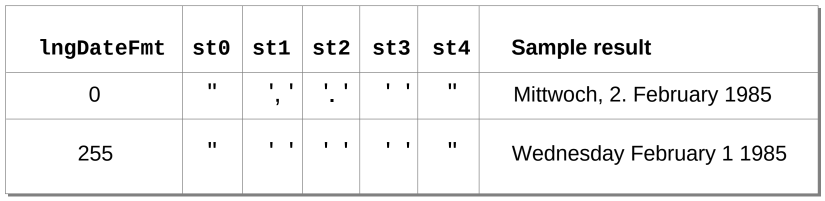
Table B-5 Separator positions in long date format

`lngDateFmt`	**Long date format**				
0	`st0` *day name*	`st1` *day*	`st2` *month*	`st3` *year*	`st4`
255	`st0` *day name*	`st1` *month*	`st2` *day*	`st3` *year*	`st4`

`intl1Vers`	Region code and version number. The code number of the region that this resource applies to is in the high-order byte, and the version number of this long-date-format resource is in the low-order byte.
`localRtn`	Originally designed to contain a routine that localizes sorting order; now unused for that purpose. If an extended long-date-format resource is available (see the next section), this field contains the hexadecimal value $A89F as the first word.

Figure B-3 gives two examples of how the Text Utilities routines format dates based on the fields in the numeric-format resource. The examples assume that the `suppressDay` and `dayLeading0` fields contain 0. If the `SuppressDay` field contains a value of 255, the formatting routines omit the day and the punctuation indicated in the `st1` field. If the `dayLeading0` field contains a value of 255, the Text Utilities place a 0 before day numbers less than 10.

Figure B-3 Examples of long date formatting



`lngDateFmt`	**`st0`**	**`st1`**	**`st2`**	**`st3`**	**`st4`**	**Sample result**
0	"	','	'.'	' '	"	Mittwoch, 2. February 1985
255	"	' '	' '	' '	"	Wednesday February 1 1985

The Itl1ExtRec Data Type

The standard long-date-format resource has several limitations. First, it assumes that seven day names and 12 month names are sufficient, which is not true for some calendars. For example, the Jewish calendar can have 13 months in some years. Second, it assumes that day and month names can be abbreviated by simply truncating them to a fixed length, but this not true in many languages.

An optional extension to the long-date-format resource provides additional information that solves these problems. The Text Utilities routines that generate date strings use information in the `'itl1'` extension if it is present.

The standard long-date-format resource ends with a variable-length field (`localRtn`) originally intended to be used for code that alters the built-in U.S. sorting behavior. This field is no longer needed, because code for changing the sorting behavior is now in the string-manipulation (`'itl2'`) resource.

In existing unextended long-date-format resources, the `localRtn` field contains a single `RTS` instruction (hexadecimal $4E75). In extended long-date-format resources, the hexadecimal value $A89F is the first word in the`localRtn` field. (This is the unimplemented trap instruction, which could not have been the first word of any valid local routine.) The resource extension follows immediately after that word.

You can access the contents of the `'itl1'` resource extension through the `Itl1ExtRec` data type.

```
TYPE Itl1ExtRec =
RECORD
   base:                    Intl1Rec;{un-extended Intl1Rec}
   version:                 Integer; {version number}
   format:                  Integer; {format code}
   calendarCode:            Integer; {calendar code for 'itl1'}
   extraDaysTableOffset:    LongInt; {offset to extra days table}
   extraDaysTableLength:    LongInt; {length of extra days table}
   extraMonthsTableOffset:  LongInt; {offset to extra months table}
   extraMonthsTableLength:  LongInt; {length of extra months table}
   abbrevDaysTableOffset:   LongInt; {offset to abbrev. days table}
   abbrevDaysTableLength:   LongInt; {length of abbrev. days table}
   abbrevMonthsTableOffset:LongInt; {offset to abbr. months table}
   abbrevMonthsTableLength:LongInt; {length of abbr. months table}
   extraSepsTableOffset:    LongInt; {offset to extra seps table}
   extraSepsTableLength:    LongInt; {length of extra seps table}
   tables:                  ARRAY[0..0] OF Integer;
                                     {the tables; variable-length}
END;
```

Field descriptions

`base`	A standard (unextended) long-date-format resource.
`version`	The version number of this extension. Unlike the `intl1Vers` field in the unextended `'itl1'` resource, this field contains nothing but the version number.

`format` A number that identifies the format of this resource. The current extended long-date-format resource format has a format code of 0.

`calendarCode` Multiple calendars may be available on some systems, and it is necessary to identify the particular calendar for use with this long-date-format resource. Constants for the currently defined calendars are as follows:

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

The Script Manager initializes part of the script variable accessed through the selector `smScriptNumDate` with the value in this field.

`extraDaysTableOffset`
The offset from the beginning of the long-date-format resource to the extra days table.

`extraDaysTableLength`
The length in bytes of the extra days table.

`extraMonthsTableOffset`
The offset from the beginning of the long-date-format resource to the extra months table.

`extraMonthsTableLength`
The length in bytes of the extra months table.

`abbrevDaysTableOffset`
The offset from the beginning of the long-date-format resource to the abbreviated days table.

`abbrevDaysTableLength`
The length in bytes of the abbreviated days table.

`abbrevMonthsTableOffset`
The offset from the beginning of the long-date-format resource to the abbreviated months table.

`abbrevMonthsTableLength`
The length in bytes of the abbreviated months table.

`extraSepsTableOffset`
The offset from the beginning of the long-date-format resource to the extra separators table.

`extraSepsTableLength`
The length in bytes of the extra separators table.

`tables` The tables that make up the rest of the `'itl1'` resource extension.

Each table in the `'itl1'` resource extension is an array consisting of an integer count followed by a list of Pascal strings specifying names of days, names of months, or separators.

- Extra days table. A list of names. This format is for those calendars with more than 7 day names.
- Extra months table. A list of names. This format is for those calendars with more than 12 months.
- Abbreviated days table. A table of abbreviations. If the header specifies an offset to and length of the abbreviated days table, the Text Utilities routines that create date strings use this table instead of truncating day names to the number of characters specified in the `abbrevLen` field of the standard `'itl1'` resource.
- Abbreviated months table. A table of abbreviations. If the header specifies an offset to and length of the abbreviated months table, the Text Utilities routines that create date strings use this table instead of truncating month names to the number of characters specified in the `abbrevLen` field of the standard `'itl1'` resource.
- Extra separators table. A list of additional date separators. When parsing date strings, the Text Utilities `StringToDate` and `StringToTime` functions permit the separators in this list to be used in addition to the date separators specified elsewhere in the numeric-format and long-date-format resources.

String-Manipulation Resource (Type 'itl2')

The string-manipulation resource (resource type `'itl2'`) is used by the Text Utilities and the Script Manager for defining and comparing text elements.

The string-manipulation resource contains routines, called sorting hooks, that perform string sorting; it also contains tables that define character type, case conversion, and word boundaries. The Text Utilities routines `IdenticalString`, `IdenticalText`, `CompareString`, and `CompareText` call the Text Utilities sorting hooks. The Text Utilities routines `CharacterType`, `FindWordBreaks`, `LowercaseText`, `UppercaseText`, `StripDiacritics`, and `UppercaseStripDiacritics`, and the Script Manager function `TransliterateText`, may make use of tables in the string-manipulation resource.

By replacing the sorting hooks, you can modify the way string comparisons are made; by replacing tables in the string-manipulation resource, you can modify how word boundaries are determined. See the sections “Supplying Custom Sorting Routines” on page B-43, and “Supplying Custom Word-Break Tables” on page B-44, for more information.

Each enabled script system has one or more string-manipulation resources. The resource ID for each one is within the range of resource ID numbers for that script system. The default `'itl2'` resource for a script is specified in the `itlbSort` field of the script's international bundle resource (type `'itlb'`).

Note
The resource template used by `Rez` and `DeRez` specifies a particular ordering of code and tables in the string-manipulation resource, although that order is not required for the resource to be used correctly. ◆

Each string-manipulation resource contains the following elements:

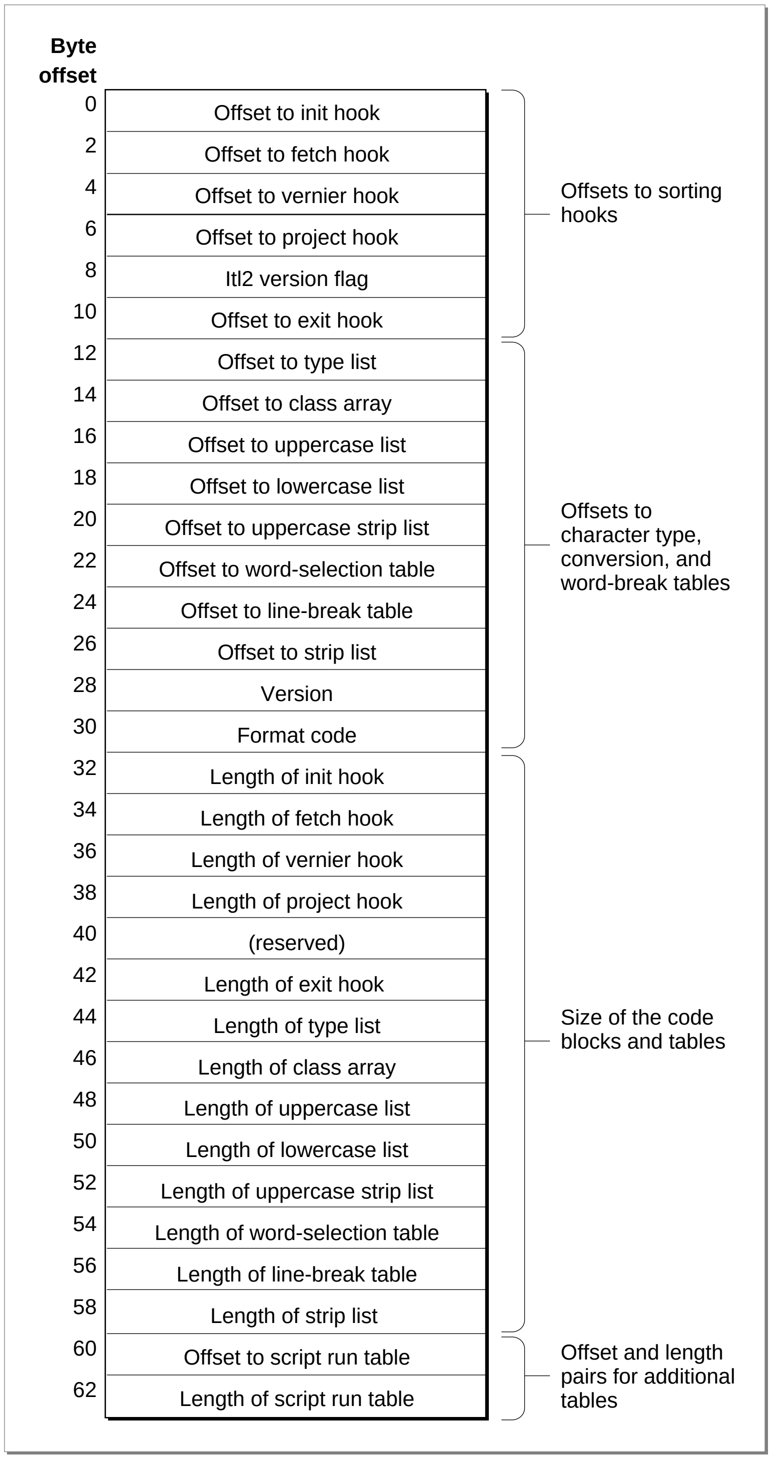
- header
- string comparison routines (sorting hooks)
- character-type tables (optional)
- case-conversion and stripping tables (optional)
- word break tables
- subscript table (optional)

The string-manipulation resource described in this section is sometimes called the *extended* `'itl2'` *resource*. Prior to Macintosh system software version 6.0.4, a more abbreviated `'itl2'` version was supported. That original`'itl2'` resource consisted of the header and sorting hooks only. The full resource as documented here is supported by system software versions 7.0 and later.

Resource Header

The string-manipulation resource header allows you to access the code segments and tables that make up the resource. All fields in the header are 16-bit words. Each field designated as an offset contains the signed offset, in bytes, from the beginning of the resource to the specified code block or table. The header is followed by the actual code segments and tables, which may be in any order. Figure B-4 shows the structure of the header.

Figure B-4 Format of the string-manipulation resource header


Byte offset
0 Offset to init hook
2 Offset to fetch hook
4 Offset to vernier hook
6 Offset to project hook
8 Itl2 version flag
10 Offset to exit hook
12 Offset to type list
14 Offset to class array
16 Offset to uppercase list
18 Offset to lowercase list
20 Offset to uppercase strip list
22 Offset to word-selection table
24 Offset to line-break table
26 Offset to strip list
28 Version
30 Format code
32 Length of init hook
34 Length of fetch hook
36 Length of vernier hook
38 Length of project hook
40 (reserved)
42 Length of exit hook
44 Length of type list
46 Length of class array
48 Length of uppercase list
50 Length of lowercase list
52 Length of uppercase strip list
54 Length of word-selection table
56 Length of line-break table
58 Length of strip list
60 Offset to script run table
62 Length of script run table
Offsets to sorting hooks
Offsets to character type, conversion, and word-break tables
Size of the code blocks and tables
Offset and length pairs for additional tables

The header consists of four sections. The first section contains offsets to the sorting hooks; the second section contains offsets to tables for character type, case conversion and diacritical stripping, and word break; the third section contains the lengths of all of the code blocks and tables; the fourth section contains offset and length pairs for tables to be added in the future.

- The first section of the header contains a version flag and five offsets to the sorting hooks: init hook, fetch hook, vernier hook, project hook, and exit hook. The sorting hooks are string-comparison routines, code segments that control sorting behavior. The hooks can replace or modify the built-in U.S. sorting behavior, on a character-by-character basis.

  The Itl2 version flag is a long integer value that describes the format of this string-manipulation resource. A value of –1 indicates that this string-manipulation resource is in the system software version 6.0.4 (or newer) format. In versions previous to 6.0.4, this element contains the offset to the reserved hook, another sorting hook. In versions previous to 6.0.4, the string-manipulation resource header stops at this point.
- The second section of the resource header contains the following elements:
  - Offsets to the character-type tables: type list, class array.
  - Offsets to the case conversion and diacritical stripping lists: uppercase list, lowercase list, uppercase strip list, strip list.
  - Offsets to the word-break tables: word-selection table, line-break table.
  - Version number. The version number of this string-manipulation resource.
  - Format code. Contains 0 if the string-manipulation resource header stops at this point (true for system software version 6.0.4); contains 1 if the string-manipulation resource header has the format shown in Figure B-4 (true for system software version 7.0 and later).
- The third section of the header contains the lengths of all of the code blocks and tables for which there are offsets in the first two sections. The Script Manager requires valid length values in this section only for those tables that can be accessed through the `GetIntlResourceTable` procedure (the word-selection and line-break tables).
- The fourth section contains offset and length pairs for additional tables. The first pair in this section is used for an optional table (`findScriptTable`) defining characters of a subscript within a non-Roman script system. It is used by the Script Manager `FindScriptRun` function. See “Script Run Table Format” beginning on page B-40. If this table is not present, the offset and length are 0.

The 'itl2' Sorting Hooks

The string-manipulation resource contains five sorting hooks, each of which can modify the functioning of its equivalent default sorting routine that is built into Text Utilities. If the sorting hooks are all empty, the default U.S. Roman sorting behavior results. For example, the `'itl2'` resource in the version of the Roman script system that has been localized for the United States contains the built-in sorting behavior and empty hooks. For other script systems, one or more of the hooks are replaced with actual routines, to handle characters that need to be sorted differently from the default—for example, the

Spanish character combination “rr” or the Norwegian “ñ”. Most of the sorting routines are called in turn for each character in each string of a pair that are being compared. Here is what each of the routines does:

- Init routine. The init routine prepares two strings for comparison. The Text Utilities sorting routines compare a pair of strings byte for byte, and pass control to the init routine as soon as a pair of unequal byte values occur. All the init routine does is check to see if either of the byte values is the second byte of a 2-byte character (or other sorting unit, such as “rr” in Spanish). If it is, the init routine backs up one byte in the string, and passes control to the fetch routine.
- Fetch routine. The fetch routine fetches the next sorting unit from each string, taking into account whether the unit is composed of one or two bytes. Many, though not all, characters in 2-byte script systems are 2 bytes long. Character combinations in 1-byte scripts can also be considered as single sorting units during sorting—such as “ch” in Spanish and “dz” in Croatian. For example, consider the second characters in these two strings:

  b c h a
  b c a d

  In analyzing the second sorting unit of each string, English versions of the fetch routine would return “c” in each case. Spanish versions, which combine “c” and “h” into a singular sorting unit “ch”, would return “ch” for the upper string and “c” for the lower string.
- Project routine. The project routine defines the primary sorting position for the individual sorting unit passed to it. In the example just presented, the English version of the project routine would give the same result for the second sorting unit of each string, whereas the Spanish version would give them different values.

  Where secondary sorting exists, the project routine “projects” each character into the sorting position of its equivalent primary character (perhaps uppercase with diacritical marks stripped). For example, consider the following two strings:

  b C a d f
  B c å d g

  The project routine would give identical results for all the character pairs until passed the “f” and “g”. In terms of the project routine, the strings would be sorted as if they were

  B C A D F
  B C A D G

  The Text Utilities use the project routine to establish decision characters to be used later if a primary difference is not available. The first pair of sorting units that have the same projected position but are not byte-for-byte identical is saved from this. Those decision characters are acted upon by the vernier routine.
- Vernier routine. The vernier routine is the tie breaker that determines the sorting order for strings that are equivalent in terms of primary sorting. It defines the secondary sorting position for the sorting unit passed to it. Suppose, in the previous example, that the strings were

  b c a d f
  b c å d f

Primary equivalence exists between the two strings. The decision characters “a” and “å” are passed in turn to the vernier routine; the vernier routine passes back a sorting position for each one. The return values determine whether “a” sorts before or after “å”, and thus establish the sorting order for the strings.

- Exit routine. This sorting hook exists to allow for any needed post-processing after the sorting order for a pair of strings has been determined. It is called just before the Text Utilities string-comparison routine returns to the caller.

For information on providing custom versions of the sorting hooks, see “Supplying Custom Sorting Routines” on page B-43.

The 'itl2' Tables

The following tables in the string-manipulation resource define character and word features for processing strings.

- Type list. Contains character-type information for each class of character (as specified by the class array) in the script system’s character set. The Script Manager `CharacterType` function uses this table. The type list is used by 1-byte script systems only; character-type information for a 2-byte script system is in that script’s encoding/rendering (`'itl5'`) resource.
- Class array. Maps each character in the script system’s character set to a *class,* which is used to index into the other character tables in the string-manipulation resource. The Script Manager `CharacterType` function uses this table. The class array is used by 1-byte script systems only; character-class information for a 2-byte script system is in that script’s encoding/rendering (`'itl5'`) resource.
- Uppercase list. Used to generate uppercase equivalents for all lowercase characters in the script system’s character set. For each character class, contains a value to be added to the character code to convert all characters to uppercase. The Text Utilities `UppercaseText` procedure uses this table. The uppercase list is used by 1-byte script systems only.
- Lowercase list. Used to generate lowercase equivalents for all uppercase characters in the script system’s character set. For each character class, contains a value to be added to the character code to convert all characters to lowercase. The Text Utilities `LowercaseText` procedure uses this table. The lowercase list is used by 1-byte script systems only.
- Uppercase strip list. Used to generate uppercase equivalents—without diacritical marks—for all characters in the script system’s character set. For each character class, contains a value to be added to the character code to convert all characters to uppercase versions without diacritical marks. The Text Utilities `UppercaseStripDiacritics` procedure uses this table. The uppercase strip list is used by 1-byte script systems only.
- Strip list. Used to generate equivalents—without diacritical marks—for all characters in the script system’s character set. For each character class, contains a value to be added to the character code to strip diacritical marks. The Text Utilities `StripDiacritics` procedure uses this table. The strip list is used by 1-byte script systems only.

- Word-selection table. A table of data type `NBreakTable` or `BreakTable`, used by the Text Utilities `FindWordBreaks` procedure to find word boundaries for the purposes of word selection. See "Supplying Custom Word-Break Tables" on page B-44 for a description of the break-table formats.
- Line-break table. A table of data type `NBreakTable` or `BreakTable`, used by theText Utilities `FindWordBreaks` procedure to find word boundaries for breaking lines of text. The rules governing word boundaries for line breaking are generally somewhat different from those for word selection.
- Script run table. A data structure used by the Text Utilities `FindScriptRun` function. It is used to find runs of a subscript, such as Roman, within text of a non-Roman script system. See the next section.

Script Run Table Format

The script run table is used by the Text Utilities `FindScriptRun` function. `FindScriptRun` locates runs of text that belong to a subscript, such as Roman, within a single script run. The `FindScriptRun` function is described in the chapter "Text Utilities" in this book.

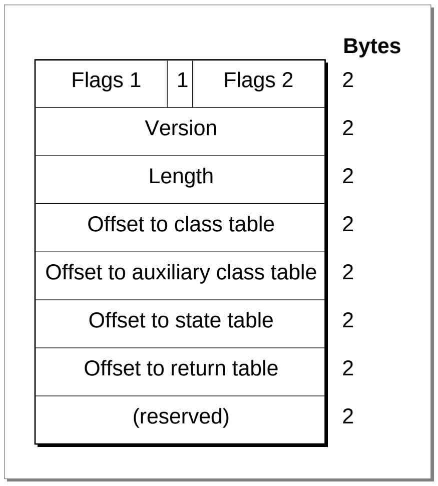
There are two formats of script run table. The original format, used in versions of system software earlier than 7.1, consists of a series of byte pairs with the format *character code, script code*. The character code is the final character code in a range of characters that belongs to the script specified by the script code. (The table contains only final character codes; the initial character code of a range is assumed to be one greater than the final character code in the previous range—or 0 for the first range.) The last pair must have character code $FF. For example, if the character set encoding for script `smSample` were defined such that $00–7F and $A0 were Roman characters and the remaining characters were native characters in `smSample`, the table would appear as follows:

Character code	Script code
$7F	`smRoman`
$9F	`smSample`
$A0	`smRoman`
$FF	`smSample`

This simple format is appropriate for script systems whose text can be separated into Roman or native characters based purely upon character code, and for which other subscript information (returned in the `variant` field of the `ScriptRunStatus` record by `FindScriptRun`) is always 0. For 2-byte script systems, or when the same character could be designated as either Roman or native (depending on its context), this simple format is insufficient.

The newer format for the script run table is used in versions of system software starting with 7.1. It consists of a header, a state table, and a set of associated tables, similar in structure to the word-break table of type `NBreakTable` (described on page B-44). It is more flexible than the old format: for example, it can consider punctuation marks such as the period (ASCII code $2E) to be either to Roman or non-Roman, depending upon whether they are associated with Roman or non-Roman characters in the text. The script run table format is shown in Figure B-5.

Figure B-5 Format of the script run table header (new format)



The table header has these elements:

- Flags 1 and flags 2. The flags are not defined and should be 0, except that the high-order bit of the second byte (flags 2) must be 1 to mark this as a new-format script run table.
- Version. The version number of this script run table format.
- Length. The length in bytes of this script run table.
- Offset to class table. The offset in bytes from the beginning of the script run table to the beginning of the class table.
- Offset to auxiliary class table. The offset in bytes from the beginning of the script run table to the beginning of the auxiliary class table.
- Offset to state table. The offset in bytes from the beginning of the script run table to the beginning of the state table.
- Offset to return table. The offset in bytes from the beginning of the script run table to the beginning of the return table.
- (reserved). Reserved.

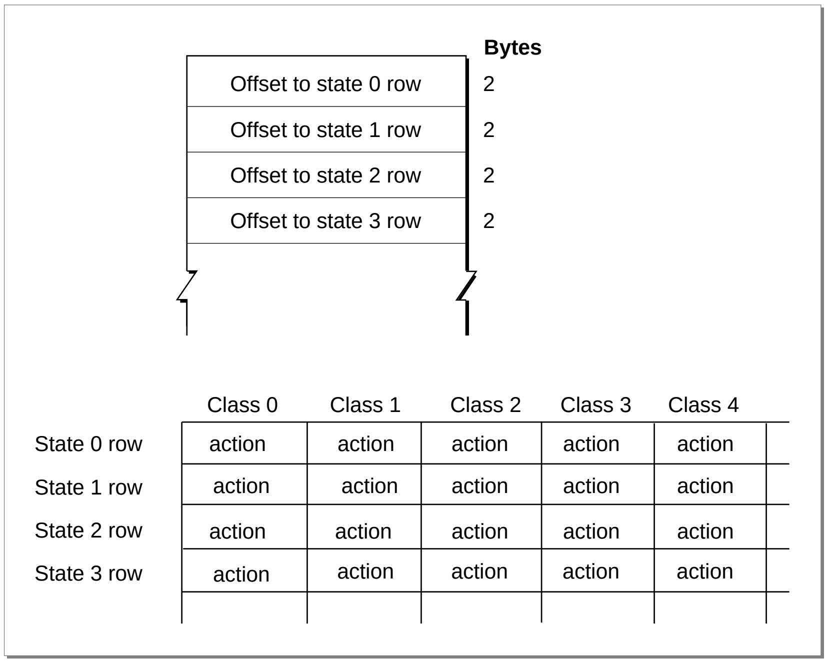
The header is immediately followed by the data of the class table, auxiliary class table, state table, and return table. The tables have this format and content:

- The class table is an array of 256 signed bytes. It assigns class values to 1-byte characters and works with the auxiliary class table to assign class values for 2-byte characters. It has the same format as the class table used by the word-break table described under "NBreakTable Format" beginning on page B-44.

- The auxiliary class table assigns character classes to 2-byte characters. It has the same format as the auxiliary class table used by the word-break table described under "NBreakTable Format" beginning on page B-44.
- The state table is used by `FindScriptRun` to determine the subscript assignment for a given character class, accounting for its context. Using the state table, `FindScriptRun` starts at a specified character, moving forward through the text until it encounters a subscript boundary.

The state table is shown in Figure B-6. The table begins with a list of words containing byte offsets from the beginning of the state table to the rows of the state table; this is followed by a *c*-by-*s* byte array, where *c* is the number of classes (columns) and *s* is the number of states (rows). The bytes in this array are stored with the column index varying most rapidly—that is, the bytes for the state 0 row precede the bytes for the state 1 row. There is a maximum of 128 classes and 64 states (including the start and exit states).

Figure B-6 Script run table state table

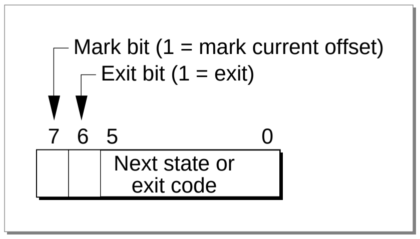


Each entry in this array is an action code, which specifies

- whether to mark the current offset
- whether to exit
- the next state or (if exiting) the exit code

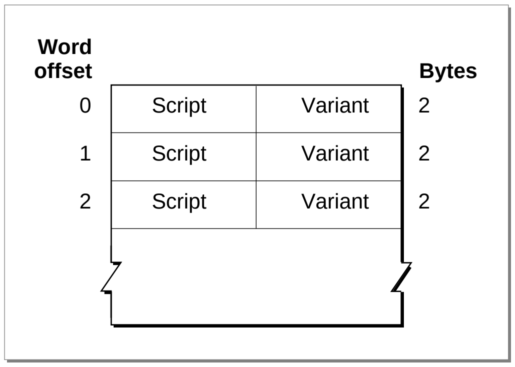
The format of an action code is shown in Figure B-7.

Figure B-7 Format of a script run table action code



The return table is a list of script code–variant pairs, as shown in Figure B-8. The table lists possible return values for the `FindScriptRun` function. Each pair in the table is a `ScriptRunStatus` record, as described in the chapter "Text Utilities" in this book. The variant associated with each script code gives subscript information for 2-byte script systems. When `FindScriptRun` exits the state table, it has encountered a subscript boundary; it uses the exit code to index into the return table and determine the script code of the subscript run it has just exited from.

Figure B-8 Format of the script run table return table



Supplying Custom Sorting Routines

String comparison in a given script system is controlled by routines accessed through the sorting hooks in the string-manipulation resource. However, there is also a default sorting behavior built into the Text Utilities, and the sorting hooks are designed to allow a given script system to use, modify, or replace parts of that default behavior, on a character-by-character basis. The U.S. version of the Roman script system, for example, uses the built-in sorting behavior exclusively; its `'itl2'` resource has only nonfunctional (empty) sorting hooks. The built-in sorting behavior used by the Text Utilities is described in the appendix "Built-in Script Support" in this book.

The sorting hooks are described under "The 'itl2' Sorting Hooks" on page B-37. You can supply a replacement string-manipulation resource with nonempty versions of any sorting hooks, to create sorting behavior more appropriate for your target region.

Note
Even "empty" sorting hooks cannot be completely empty. All `'itl2'` sorting hooks are responsible for setting the condition codes that indicate whether or not they have taken any sorting action. If these condition codes have not been set, the Text Utilities uses the default Roman sorting behavior. ◆

For more information on replacing any or all of the sorting hooks, see Macintosh Technical Note #178, available from Macintosh Developer Technical Support.

Supplying Custom Word-Break Tables

The Text Utilities `FindWordBreaks` procedure uses state machines and associated tables in a script's string-manipulation resource to determine word boundaries and line breaks.

The `FindWordBreaks` procedure examines a block of text to determine the boundaries of the word that includes a specified character in the block. Usually, `FindWordBreaks` uses different state tables to define words for word selection than it does for line breaking.

To replace the word-selection criteria, you can supply a replacement string-manipulation resource with a modified break table. This section describes the break table and how `FindWordBreaks` uses it.

NBreakTable Format

`FindWordBreaks` uses word-break tables of type `NBreakTable`, defined for system software version 7.0 and later:

```
TYPE     NBreakTable =
   RECORD
      flags1:           SignedByte; {break table format flags}
      flags2:           SignedByte; {break table format flags}
      version:          Integer;    {version no. of break table}
      classTableOff:    Integer;    {offset to ClassTable array}
      auxCTableOff:     Integer;    {offset to auxCTable array}
      backwdTableOff:   Integer;    {offset to backwdTable array}
      forwdTableOff:    Integer;    {offset to forwdTable array}
      doBackup:         Integer;    {skip backward processing?}
      length:           Integer;    {length of the break table}
      charTypes:        ARRAY[0..255] OF SignedByte;
      tables:           ARRAY[0..0] OF Integer;
                                    {break tables}
END;

TYPE NBreakTablePtr = ^NBreakTable;
```

Field descriptions

`flags1`	The high-order byte of the break table format flags. If the high-order bit of this byte is set to 1, this break table is in the format used by `FindWordBreaks`.
`flags2`	The low-order byte of the break table format flags. If the value in this byte is 0, the break table belongs to a 1-byte script system; in this case `FindWordBreaks` does not check for 2-byte characters.
`version`	The version number of this break table.
`classTableOff`	The offset in bytes from the beginning of the break table to the beginning of the class table.
`auxCTableOff`	The offset in bytes from the beginning of the break table to the beginning of the auxiliary class table.
`backwdTableOff`	The offset in bytes from the beginning of the break table to the beginning of the backward-processing table.
`forwdTableOff`	The offset in bytes from the beginning of the break table to the beginning of the forward-processing table.
`doBackup`	The minimum byte offset into the buffer for doing backward processing. If the selected character for `FindWordBreaks` has a byte offset less than `doBackup`, `FindWordBreaks` skips backward processsing altogether and starts from the beginning of the buffer.
`length`	The length in bytes of the entire break table, including all the individual tables.
`charTypes`	The class table. See explanation below.
`tables`	The data of the auxiliary class table, backward table, and forward table.

The tables have this format and content:

- The class table is an array of 256 signed bytes. Offsets into the table represent byte values; if the entry at a given offset in the table is positive, it means that a byte whose value equals that offset is a single-byte character, and the entry at that offset is the class number for the character. If the entry is negative, it means that the byte is the first byte of a 2-byte character code, and the auxiliary class table must be used to determine the character class. Odd negative numbers are handled differently from even negative numbers.
- The auxiliary class table assigns character classes to 2-byte characters. It is used when the class table determines that a byte value represents the first byte of a 2-byte character.

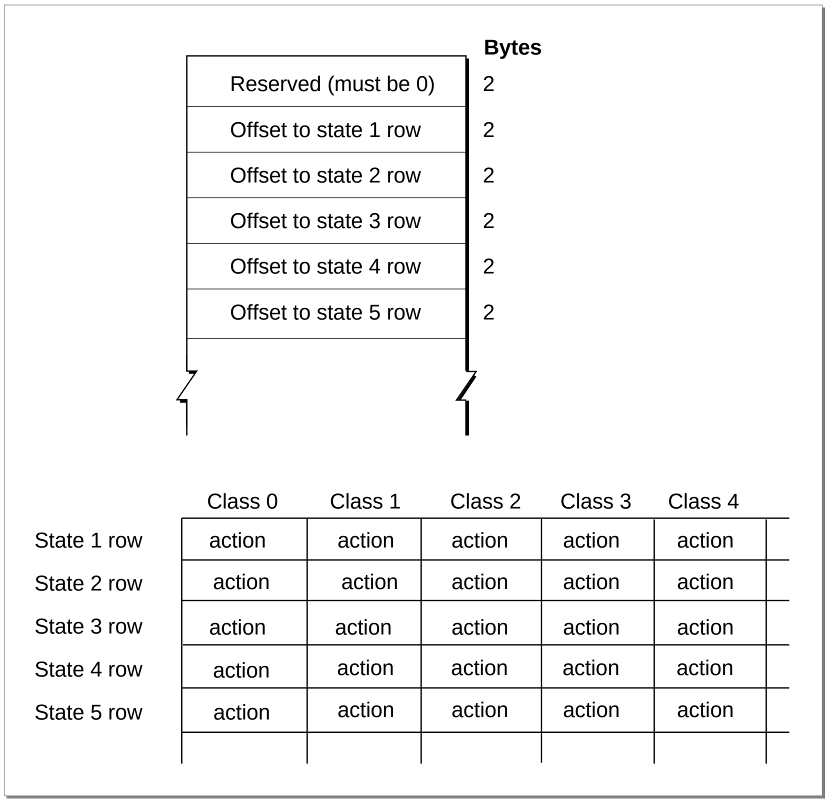
  - □ Here is how the auxiliary class table handles odd negative values from the class table. If the first word of the auxiliary class table is equal to or greater than zero, it represents the default class number for 2-byte character codes—the class assigned to every odd negative value from the class table. If the first word is less than zero, it is the negative of the offset from the beginning of the auxiliary class table to a first-byte class table (a table of 2-byte character classes that can be determined from just the first byte). The value from the class table is negated, 1 is subtracted from it to obtain an even offset, and the value at that offset into the first-byte class table is the class to be assigned.
  - □ Here is how the auxiliary class table handles even negative values from the class table. The auxiliary class table begins with a variable-length word array. The words that follow the first word are offsets to row tables. Row tables have the same format as the class table, but are used to map the second byte of a 2-byte character code to a class number. If the entry in the class table for a given byte is an even negative number, `FindWordBreaks` negates this value to obtain the offset from the beginning of the auxiliary class table to the appropriate word, which in turn contains an offset to the appropriate row table. That row table is then used to map the second byte of the character to a class number.

- ■ The backward-processing table is a state table used by `FindWordBreaks` for backward searching. Using the backward-processing table, `FindWordBreaks` starts at a specified character, moving backward as necessary until it encounters a word boundary.
- ■ The forward-processing table is a state table used by `FindWordBreaks` for forward searching. Using the forward-processing table, `FindWordBreaks` starts at one word boundary and moves forward until it encounters another word boundary.

The backward-processing table and the forward-processing table have the same format, as shown in Figure B-9. The table begins with a list of words containing byte offsets from the beginning of the state table to the rows of the state table; this is followed by a *c*-by-*s* byte array, where *c* is the number of classes (columns) and *s* is the number of states (rows). The bytes in this array are stored with the column index varying most rapidly—that is, the bytes for the state 1 row precede the bytes for the state 2 row.

Note
There is a maximum of 128 classes and 64 states (including the start and exit states). ◆

Figure B-9 `NBreakTable` state table



Each entry in this array is an action code, which specifies

- whether to mark the current offset
- the next state, which may be the exit state (state 0)

The format of an action code is shown in Figure B-10.

Figure B-10 Format of an `NBreakTable` action code

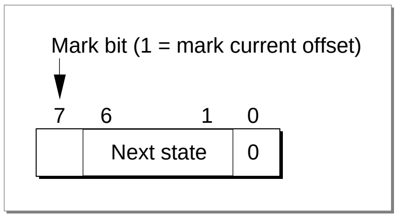


Table B-6 shows an example of the classes used in a state table. It is taken from the word-selection table of the U.S. localized version of the Roman script system.

Table B-6 Example of classes for an `NBreakTable` state table

Class number	Class name	Used for
0	break	Everything not included below
1	nonBreak	Nonbreaking spaces
2	letter	Letters, ligatures, and accents
3	number	Digits
4	midLetter	Hyphen
5	midLetNum	Apostrophe (vertical or right single quote)
6	preNum	$ £ ¥ ¤
7	postNum	% ‰ ¢
8	midNum	,/
9	preMidNum	.
10	blank	Space, tab, null
11	cr	Return

Table B-7 shows an example of the defined states for a state table. It is taken from the forward-processing table of the word-selection table of the U.S. localized version of the Roman script system.

Table B-7 Example of states for an `NBreakTable` state table

State number	Explanation
0	Exit
1	Start, or has detected initial *nonBreak* sequence
2	Has detected a *letter*
3	Has detected a *number*
4	Has detected a non-whitespace character that should stand alone; now anything but *nonBreak* generates an exit
5	Has detected *preMidNum* or *preNum*; now anything but *number* or *nonBreak* generates an exit
6	Has detected a *blank*

Table B-7 Example of states for an `NBreakTable` state table (continued)

State number	Explanation
7	Has detected *letter* followed by *midLetter*, *midLetNum*, or *preMidNum*; now anything but *letter* generates an exit
8	Has detected a non-whitespace character followed by *nonBreak* (the *nonBreak* should be treated as non-whitespace)
9	Has detected *number* followed by *midNum*, *midLetNum*, or *preMidNum*; now anything but *number* generates an exit
10	Marks current offset, then exits
11	Has detected *blank* followed by *nonBreak* (the *nonBreak* should be treated as a blank)

How FindWordBreaks Uses the Break Table

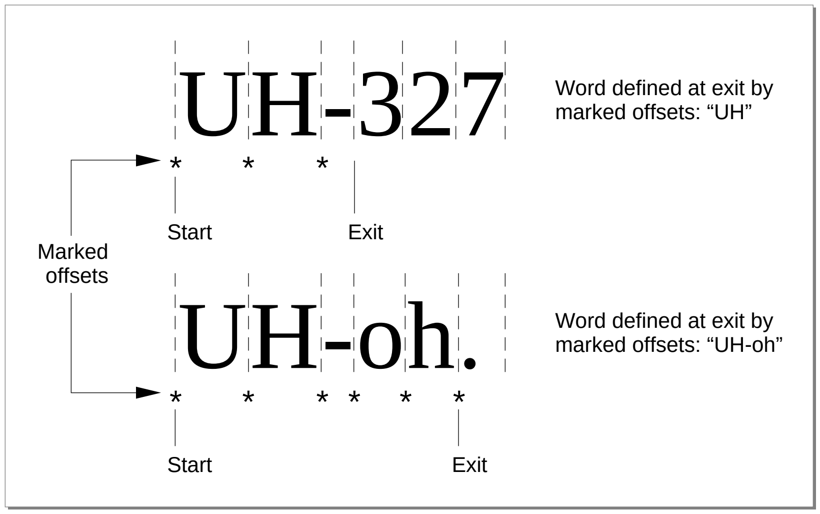
`FindWordBreaks` uses a state machine to determine the word boundaries on either side of a given character in a text buffer. The state machine must start at a point in the buffer at or before the beginning of the word that includes that character. If the specified character is sufficiently close to the beginning of the text buffer (controlled by the `doBackupMin` parameter in the break table), the state machine simply starts from the beginning of the buffer. Otherwise, `FindWordBreaks` uses the backward-processing table to work backwards from the specified character, analyzing characters until it encounters a word boundary.

Once determined, this starting location is saved as an initial word boundary. From this point the `FindWordBreaks` state machine moves forward using the forward-processing table until it encounters another word boundary. If that word boundary is still before the specified character, its location is saved as the starting point and the state machine is restarted from that location. This process repeats until the state machine finds a word boundary that is after the specified character. At that point, `FindWordBreaks` returns the location of the previously saved word boundary and the current word boundary as the offset pair defining the word.

The state machine operates in a similar manner whether moving backward or forward; any differences in behavior are determined by the tables. The machine begins in the start state (state 1). It then cycles one character at a time until it finds a boundary break and exits. In each cycle, the current character is mapped to a class number, and the character class and the current state are used as indices into the array of action codes in the state table. Each action code specifies the next state and whether to mark the current offset. When the state machine exits, it has encountered a word boundary. The location of the word boundary is the last character offset that was marked.

Figure B-11 gives two examples of the forward operation of the state machine for word selection. It shows that an exit may or may not be generated at a hyphen, depending on the character that follows. It also shows that the marked offset on exit may or may not include the last character before the exit was generated.

Figure B-11 Forward operation of the state machine for word selection


UH-327
Word defined at exit by marked offsets: “UH”
Start
Exit
Marked offsets
UH-oh.
Word defined at exit by marked offsets: “UH-oh”
Start
Exit

Tokens Resource (Type 'itl4')

The tokens resource (resource type `'itl4'`) contains information needed to convert text into a series of language-independent tokens. Compilers, interpreters, and other expression evaluators convert source text to tokens as an initial step in their processing. The Script Manager `IntlTokenize` function uses information in the tokens resource to produce tokens from source text.

The tokens resource also contains tables for converting tokens into text, for formatting numbers, and for determining whitespace characters.

Each enabled script system has one or more tokens resources. The resource ID for each one is within the range of resource ID numbers for that script system. The default `'itl4'` resource for a script is specified in the `itlbToken` field of the script’s international bundle (`'itlb'`) resource.

Each tokens resource contains the following:

- Header
- Tokenizing tables and code
- Untoken table

- Number parts table
- Whitespace table

The tokenizing tables and code are used by the `IntlTokenize` function. The untoken table is used by the `IntlTokenize` function and by applications that want to convert tokens to strings. The number parts table is used by the Text Utilities number-formatting routines and by applications that do their own number formatting. The whitespace table is available for application use.

The NItl4Rec Data Type

The tokens resource is defined by the `NItl4Rec` data type as follows:

```
TYPE  NItl4Rec =
   RECORD
      flags:            Integer;    {reserved}
      resourceType:     LongInt;    {contains 'itl4'}
      resourceNum:      Integer;    {resource ID}
      version:          Integer;    {version number}
      format:           Integer;    {format code}
      resHeader:        Integer;    {reserved}
      resHeader2:       LongInt;    {reserved}
      numTables:        Integer;    {number of tables}

      mapOffset:        LongInt;    {offset to token table}
      strOffset:        LongInt;    {offset to string-copy rtn.}
      fetchOffset:      LongInt;    {offset to ext. fetch routine}
      unTokenOffset:    LongInt;    {offset to untoken table}
      defPartsOffset:   LongInt;    {offset to number parts table}
      whtSpListOffset:  LongInt;    {offset to whitespace table}

      resOffset7:       LongInt;    {reserved}
      resOffset8:       LongInt;    {reserved}
      resLength1:       Integer;    {reserved}
      resLength2:       Integer;    {reserved}
      resLength3:       Integer;    {reserved}
      unTokenLength:    Integer;    {length of untoken table}
      defPartsLength:   Integer;    {length of number parts table}
      whtSpListLength:  Integer;    {length of whitespace table}
      resLength7:       Integer;    {reserved}
      resLength8:       Integer;    {reserved}
   END;

TYPE NItl4Ptr     = ^NItl4Rec;
TYPE NItl4Handle  = ^NItl4Ptr;
```

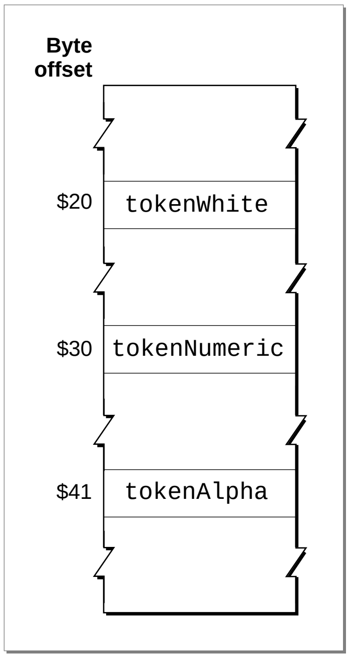
Field descriptions

`flags`	(reserved)
`resourceType`	`'itl4'` (the resource type of the tokens resource).
`resourceNum`	The resource ID number of this tokens resource.
`version`	The version number of this tokens resource.
`format`	The format code, a number that identifies the format of this tokens resource.
`resHeader`	(reserved)
`resHeader2`	(reserved)
`numTables`	The number of tables in this tokens resource.
`mapOffset`	The offset in bytes from the beginning of the resource to the token table, an array that maps each byte to a token type.
`strOffset`	The offset in bytes from the beginning of the resource to the token-string copy routine, a code segment that creates strings that correspond to the text that generated each token.
`fetchOffset`	The offset in bytes from the beginning of the resource to the extension-fetching routine, a code segment that fetches the second byte of a 2-byte character for the `IntlTokenize` function.
`unTokenOffset`	The offset in bytes from the beginning of the resource to the untoken table, an array that maps token types back to the canonical strings that represent them.
`defPartsOffset`	The offset in bytes from the beginning of the resource to the number parts table, an array of characters that correspond to each part of a number format (used primarily by the `FormatRecToString` and `StringToExtended` functions).
`whtSpListOffset`	The offset in bytes from the beginning of the resource to the whitespace table, a list of all the characters that should be treated as whitespace—for example, blank and tab for the Roman script system.
`resOffset7`	(reserved)
`resOffset8`	(reserved)
`resLength1`	(reserved)
`resLength2`	(reserved)
`resLength3`	(reserved)
`unTokenLength`	The length in bytes of the untoken table.
`defPartsLength`	The length in bytes of the number parts table.
`whtSpListLength`	The length in bytes of the whitespace table.
`resLength7`	(reserved)
`resLength8`	(reserved)

The Token Table

The `'itl4'` resource includes the token table, an array of type `mapCharTable`. The token table, also called the character-mapping table, maps each possible byte value in a 1-byte character set into a token type. Its format is shown in Figure B-12.

Figure B-12 Format of the token table



The table consists of 256 bytes. The byte offset of each location in the table represents a character code: location 0 represents a character code of 0, location 255 represents a character code of 255. Each location in the table contains a token code that represents the type of token corresponding to that character code. Constants for all defined token codes are listed in the chapter "Script Manager" in this book.

The token table is used to define tokens in 2-byte script systems also. Any location in the table that has a value of –1 represents the first byte of a 2-byte character. When it encounters such a byte, the `IntlTokenize` function calls the extension-fetching routine, described next, which analyzes that byte and the subsequent byte in the source text to determine what type of token is represented.

The Extension-Fetching Routine

The `IntlTokenize` function uses the extension-fetching routine to retrieve the second byte of a 2-byte character and determine what type of token should represent it. When `IntlTokenize` encounters a byte value in the source text that is represented by a –1 in the token table, `IntlTokenize` calls the extension-fetching routine with register A0 pointing to the second byte of the 2-byte character.

The extension-fetching routine consults internal, script-specific tables and returns the token code associated with the byte pair. `IntlTokenize` adds that token to the token list, and continues processing with the first byte following the byte pair.

The Token-String Copy Routine

When it creates a token list, the `IntlTokenize` function offers the option of also returning Pascal strings that are the normalized equivalents of the tokens it generates. `IntlTokenize` uses the token-string copy routine to create those strings and store them in a canonical format in a string list.

Canonical format means that the string-copy routine converts all numerals into standard ASCII numbers and converts the decimal separator to a period. For example, it would convert the Thai number "๒๔๘" into one token, `tokenAltNum`, with an associated Pascal string of `'248'`.

The Untoken Table

The ***untoken table*** provides a Pascal string for any type of fixed token. A ***fixed token*** is a token whose representation is unvarying, like punctuation. Alphabetic and numeric tokens are not fixed; specifying the token does not specify the string it represents.

The untoken table contains standard representations for the fixed tokens. You can use it to display the canonical format for any fixed token in the script system of the tokens resource.

The `unTokenTable` data type describes the format of the untoken table:

```
TYPE
   UntokenTable =
   RECORD
      len:        Integer;       {length of untoken table}
      lastToken:  Integer;       {maximum token code to be used}
      index:      ARRAY[0..255] OF Integer;
                                 {offsets to Pascal strings for }
                                 { tokens; last entry = lastToken}
   END;

   UntokenTablePtr         =  ^UntokenTable;
   UntokenTableHandle      =  ^UntokenTablePtr;
```

Field descriptions

`len`	The length in bytes of the untoken table.
`lastToken`	The highest token code used in this table (for range-checking).
`index`	An array of byte offsets from the beginning of the untoken table to Pascal strings—one for each possible token type—that give the canonical format for each fixed token type. The entries in the array correspond, in order, to token code values from 0 to `lastToken`. For example, the offset to the Pascal string for `tokenColonEqual` (token code = 39) is found at offset 39 in the array.

The string data directly follows the `index` array. It is a simple concatenation of Pascal strings; for example, the Pascal string for the token type `tokenColonEqual` may consist of a length byte (of value 2) followed by the characters “`:=`”.

The Number Parts Table

The **number parts table** contains standard representations for the components of numbers and numeric strings. The Text Utilities number-formatting routines `StringToExtended` and `ExtendedToString` use the number parts table, along with a number-format string created by the `StringToFormatRec` and `FormatRecToString` routines, to create number strings in localized formats.

The `NumberParts` data type defines the number parts table:

```
TYPE NumberParts =
   RECORD
      version:       Integer;            {version of this table}
      data:          ARRAY[1..31] OF WideChar;
                                         {2-byte number parts}
      pePlus:        WideCharArr;        {positive exp. notation}
      peMinus:       WideCharArr;        {negative exp. notation}
      peMinusPlus:   WideCharArr;        {neg. or pos. exp.}
      altNumTable:   WideCharArr;        {alternate digits}
      reserved: PACKED ARRAY[0..19] OF Char;
                                         {reserved}
   END;
TYPE NumberPartsPtr = ^NumberParts;
```

`version`	An integer that specifies which version of the number parts table is being used. A value of 1 specifies the first version.
`data`	An array of 31 wide characters (2 bytes each), indexed by a set of constants. Each element of the array, accessed by one of the constants, contains 1 or 2 bytes that make up that number part. (If the element contains only one 1-byte character, it is in the low-order byte and the high-order byte contains 0.) Each number part, then, may consist of one or two 1-byte characters, or a single 2-byte character.

Of the 31 allotted spaces, 15 through 31 are reserved for up to 17 unquoted characters—special literals that do not need to be enclosed in quotes in a numeric string. See the discussion of number formatting in the chapter "Text Utilities" in this book for more information.

These are the defined constants for accessing number parts in the `data` array:

Constant	Value	Explanation
`tokLeftQuote`	1	Left quote
`tokRightQuote`	2	Right quote
`tokLeadPlacer`	3	Spacing leader format marker
`tokLeader`	4	Spacing leader character
`tokNonLeader`	5	No leader format marker
`tokZeroLead`	6	Zero leader format marker
`tokPercent`	7	Percent
`tokPlusSign`	8	Plus
`tokMinusSign`	9	Minus
`tokThousands`	10	Thousands separator
	11	(reserved)
`tokSeparator`	12	List separator
`tokEscape`	13	Escape character
`tokDecPoint`	14	Decimal separator
`tokUnquoteds`	15	(first unquoted character)
		(15 through 31 reserved)
`tokMaxSymbols`	31	Maximum symbol (for range check)

IMPORTANT

Note that these constants are unrelated to the token-type constants defined for the `IntlTokenize` function. ▲

`pePlus` An array that specifies how to represent positive exponents for scientific notation. It is a wide character array, an 11-word data structure defined by the `WideCharArr` data type. It contains up to ten 1-byte or 2-byte number parts for representing positive exponents.

`peMinus` An array that specifies how to represent negative exponents for scientific notation. It is a wide character array, an 11-word array defined by the `WideCharArr` data type. It contains up to ten 1-byte or 2-byte number parts for representing negative exponents.

`peMinusPlus`	An array that specifies how to represent positive exponents for scientific notation when the format string exponent is negative. Symbols from this array can be used with the input number string to the `StringToExtended` function; they are not for use with the `StringToFormatRec` function. The array is a wide character array, an 11-word array defined by the `WideCharArr` data type. It contains up to ten 1-byte or 2-byte number parts for representing positive exponents.
`altNumTable`	A wide character array that specifies the alternate representation of numerals. The array contains ten character codes, each of which represents an alternate numeral. If the `smsfB0Digits` bit of the script-flags word is set, you should substitute the characters in this array for the character codes $30–$39 (regular ASCII numerals) in a string whose token code is `tokenAltNum` or `tokenAltReal`. Alternate numerals and the script flags word are described with the list of selectors for script variables in the chapter "Script Manager" in this book.
`reserved`	(reserved for future expansion)

The wide character (data type `WideChar`) is a format for representing a character that may be either 1 or 2 bytes long. For a 1-byte character, the high-order (first) byte in the record is 0, and the low-order (second) byte contains the character code. For a 2-byte character, the high-order byte is nonzero.

```
TYPE
   WideChar = RECORD
   CASE BOOLEAN OF
      TRUE:
         (a: PACKED ARRAY[0..1] OF Char);    {0 = high-order char}
      FALSE:
         (b: INTEGER);
   END;
```

The wide character array (data type `WideCharArr`) consists of an integer count followed by a packed array of wide characters.

```
TYPE
   WideCharArr = RECORD
      size: INTEGER;                         {no. of entries -1}
      data: PACKED ARRAY[0..9] OF WideChar;
   END;
```

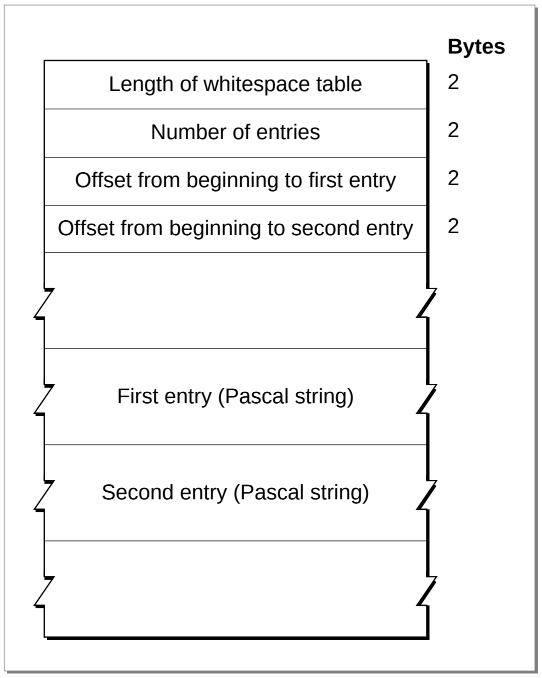
Field descriptions

`size`	The number of items in the table minus 1.
`data`	Up to ten wide characters. If the number part is only a single 1-byte character, that character is in the low-order byte of the word.

The Whitespace Table

The whitespace table contains characters that may be used to indicate white space, such as blanks, tabs, and carriage returns. Figure B-13 shows the format of the whitespace table. Each entry pointed to by the table is a Pascal string specifying a single whitespace character (which may be 1 or 2 bytes long). The strings immediately follow the offset fields.

Figure B-13 Format of the whitespace table


Bytes
Length of whitespace table
2
Number of entries
2
Offset from beginning to first entry
2
Offset from beginning to second entry
2
First entry (Pascal string)
Second entry (Pascal string)

Encoding/Rendering Resource (Type 'itl5')

The encoding/rendering resource (resource type `'itl5'`) specifies character encoding or display behavior in a given script system. The resource has different formats and functions for 1-byte and 2-byte script systems. In 1-byte script systems, it specifies character rendering behavior. In 2-byte script systems, it contains byte-type and character-type information.

The encoding/rendering resource is optional; it does not exist for all script systems. The Roman script system does not include an `'itl5'` resource.

The resource ID of an encoding/rendering resource is within the range of resource ID numbers for that script system. Although more than one encoding/rendering resource may be associated with a given script system, the script by default uses the resource specified in the `itlbEncoding` field of its international bundle (`'itlb'`) resource.

Resource Header

The header for the encoding/rendering resource is the same for 1-byte and 2-byte script systems. Its format is general enough to allow new tables to be added in the future. This is the definition of the resource-header format:

```
TYPE  Itl5Record =
   RECORD
      versionNumber:    Fixed;
      numberOfTables:   Integer;
      reserved:         ARRAY[0..2] OF Integer;
      tableDirectory:   ARRAY[0..0] OF TableDirectoryRecord;
   END;
```

Field descriptions

`versionNumber` The version of this `'itl5'` resource.

`numberOfTables`
The number of tables this resource contains.

`reserved` (for internal use)

`tableDirectory`
A directory of all the tables in the resource. Each entry in the directory is a table directory record, with this format:

```
TYPE  TableDirectoryRecord =
   RECORD
      tableSignature:   OSType ;
      reserved:         LongInt;
      tableStartOffset: LongInt;
      tableSize:        LongInt;
   END;
```

Field descriptions

`tableSignature`
A 4-byte tag (of type `OSType`) that identifies the kind of table this record refers to.

`reserved` (for internal use)

`tableStartOffset`
The number of bytes from the beginning of the resource to the beginning of the table.

`tableSize` The length of the table, in bytes.

Tables for 1-Byte Script Systems

In 1-byte script systems, the encoding/rendering resource specifies character-rendering behavior. In general, only 1-byte complex script systems—those that work with the WorldScript I script extension—include an encoding/rendering resource. The defined table types at this time are

- Script configuration table
- Line-layout metamorphosis table
- Line-layout glyph-properties table
- Character-expansion table
- Glyph-to-character table
- Break-table directory
- `FindScriptRun` tables
- Feature-list table
- Kashida priorities table
- Reordering table

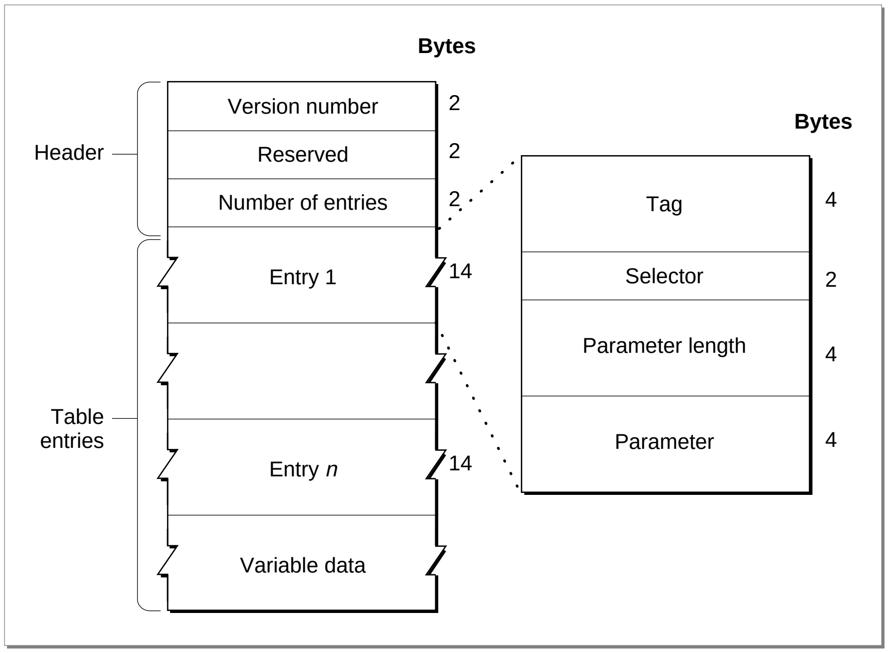
Script Configuration Table

The script configuration table (`OSType` = `'info'`) defines certain settings that affect the characteristics of a script system. The table exists so that user preferences for script configuration can be saved in a preferences file, called the script preferences file, between system restarts.

The script configuration table consists of a 6-byte header followed by a set of table entries, each of which contains a `SetScriptVariable` selector. The table entries correspond to script settings that the user can make, typically through a script-system control panel.

The format of the script configuration table is shown in Figure B-14.

Figure B-14 Format of the script configuration table



The resource header consists of three elements:

- Version number. The version number of this resource. The major version number is in the high-order byte; the minor version number is in the low-order byte.
- Reserved. A 2-byte reserved element.
- Number of entries. The number of entries in the script configuration table.

The entries immediately follow the header. Each entry has four elements:

- Tag. A 4-byte identifier of type `OSType`.
- Selector. A selector to access a script variable through the Script Manager `SetScriptVariable` function.
- Parameter length. The length of the parameter to pass to the `SetScriptVariable` function. This value always equals 4, unless this entry refers to variable-length data. See below.
- Parameter. This element contains the parameter to pass to the `SetScriptVariable` function, unless this entry refers to variable-length data. See below.

For most entries in the script configuration table, the tag is `'long'`, the parameter length is 4 (the length of a `SetScriptVariable` parameter), and the parameter element contains the `SetScriptVariable` parameter. However, a table entry may reference variable-length data, such as a string representing the name of a script system.

Such data follows the last entry in the table, and its location is specified—as an offset from the beginning of the table—in the parameter element of the table entry that references it. The data length in bytes is specified in the parameter length element of that table entry.

For example, a Hebrew encoding/rendering resource might have a script configuration table with the information shown in Table B-8.

Table B-8 A script configuration table for a Hebrew encoding/rendering resource

Offset	Value	Explanation
00	0x0100	Version number (first release = 1.0)
02	0x0000	(reserved)
04	0x0004	Four tables follow
(The table entries start here)		
06	`'long'`	The data type is a long
10	0x006	`SetScriptVariable selector smScriptRight`
12	0x0004	Four bytes follow
16	0xFFFF	–1 = right-to-left line direction
20	`'long'`	The data type is a long
24	0x0008	`SetScriptVariable` selector `smScriptJust`
26	0x0004	Four bytes follow
30	0xFFFF	–1 = right-aligned
34	`'long'`	The data type is a long
38	0x000A	`SetScriptVariable` selector `smScriptRedraw`
40	0x0004	Four bytes follow
44	0xFFFF	–1 = redraw entire line for each character
48	`'pstr'`	The data type is a Pascal string
52	0x002C	`SetScriptVariable` selector `smScriptName`
54	0x0008	The string is 8 bytes long (with length byte and pad)
58	62	Offset from beginning of table to data
62	0x6,`'Hebrew'`	The data string

In this case, the script configuration table causes the execution of four `SetScriptVariable` calls, to set the script's line direction, alignment, redraw characteristics, and name.

Each script system generally has two versions of the script configuration table: one in the encoding/rendering resource and one in a script preferences file in the Preferences folder within the user's System Folder. The table in the encoding/rendering resource has an `OSType` tag of `'info'`; the corresponding table in the preferences file is a resource of type `'CNFG'`. The script preferences file is a file of type `'pref'` with creator `'univ'`.

Both script configuration tables are used at startup. When installing a 1-byte complex script system, WorldScript I locates the script configuration table in the script's encoding/rendering resource, and loops through the table for as many times as there are entries in it, making a `SetScriptVariable` call for each entry. WorldScript I then looks for a `'CNFG'` resource for that script system in the script preferences file, and loops through that table. Thus a script system is always configured to its default settings at initialization, and then those settings are modified to reflect any user changes that have been saved. WorldScript I is described in the appendix "Built-in Script Support" in this book.

Line-Layout Metamorphosis Table

The line-layout metamorphosis table (`OSType` = `'mort'`) specifies a set of transformations that the WorldScript I contextual formatting routines can apply to the glyphs of a font. WorldScript I is described in the appendix "Built-in Script Support" in this book.

A transformation can be something simple, such as a ligature, or something complex, such as a number of changes (ligatures plus ornateness of style plus positioning of glyphs in a word). These transformations are called text features in the context of the metamorphosis table. Each text feature can have different settings, or levels of operation.

These are the text features and settings currently supported by the contextual formatting routines in WorldScript I:

- Ligature formation. Whether to form ligatures and to what extent.
- Contextual ornateness. Whether to use contextual glyphs and which set of them to use.
- Noncontextual ornateness. Which of various style and case-substitution options to use.
- Character reordering. Whether or not to reorder characters.
- Diacritical marks. Whether to show diacritical marks, hide them, or make them separate glyphs.

The line-layout metamorphosis table is identical in format to the "glyph metamorphosis table" described in the currently unpublished document *TrueType GX Font Table Formats,* available from Macintosh Developer Technical Support.

Line-Layout Glyph-Properties Table

The line-layout glyph properties table (`OSType` = `'prop'`) defines the properties associated with each glyph in a font. Examples of a glyph's properties are its line direction and whether or not it is a space character.

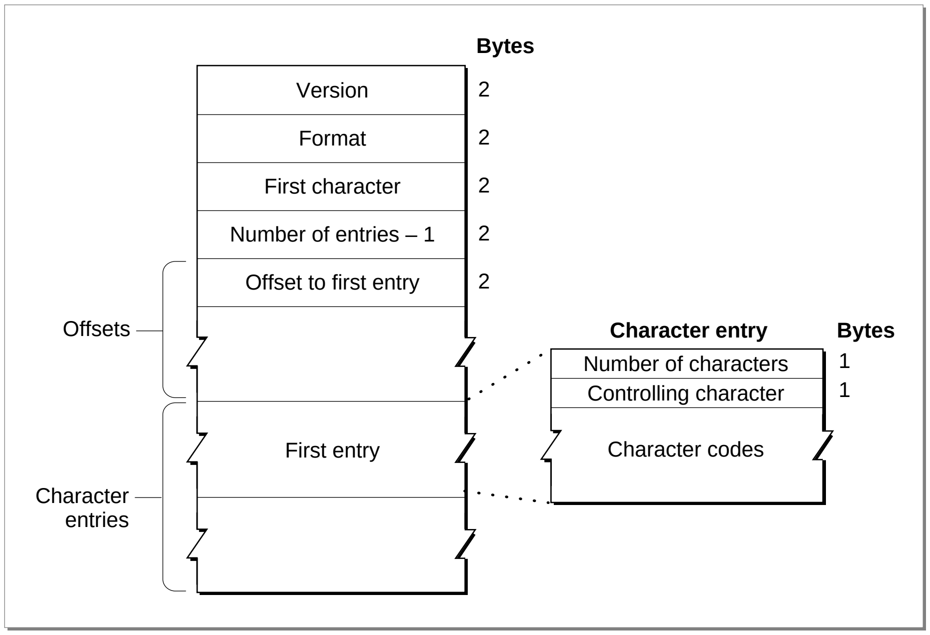
The line-layout glyph properties table is identical in format to the "glyph properties table" described in the currently unpublished document *TrueType GX Font Table Formats,* available from Macintosh Developer Technical Support.

Character Expansion Table

The character expansion table (`OSType` = `'c2c#'`) gives multiple-character equivalents to compound characters in a script system's character set. This table expands ligatures into their component characters, analogous to expanding the Roman ligature "fi" into "f" and "i". The contextual formatting routines need the character expansion table because they are specifically designed to work with a script system's fundamental character codes.

Figure B-15 shows the format of the character expansion table.

Figure B-15 Format of the character expansion table



The table has these elements:

- Version. The version number of this table. A value of $0100 means version 1.
- Format. The format code, a number that identifies the format of this table.

- First character. The character code of the first character to be expanded.
- Number of entries – 1. The number of entries in this table, as a zero-based count.
- Offsets to entries. The offset from the beginning of the table to each character entry.

The character entries immediately follow the offsets. Because the table always covers a continuous range of a character set, the character code corresponding to each character entry is calculated as (first character) + (entry number), where the first character entry is numbered 0. Each character entry has these elements:

- The total number of (expanded) characters in this entry.
- The *controlling character,* the character whose position is considered equivalent to the position of the ligature as a whole. By analogy with Roman, the controlling character in the "fi" ligature might be considered the "f", so that a mouse-down event on the leading edge of the ligature would translate, after expansion, to a mouse-down event on the leading edge of the "f".
- The character codes of the characters that are the expanded equivalent to the character code for this entry.

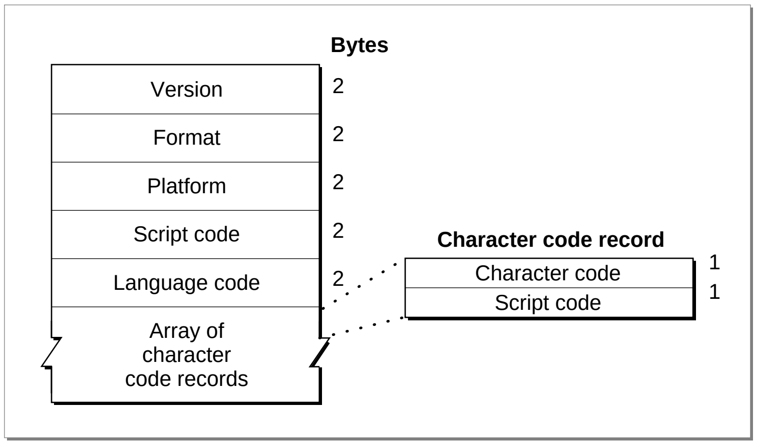
Any character within the range of character codes for this table that does not have an expanded equivalent has a value of 0 for its offset.

Glyph-to-Character Table

The glyph-to-character table (`OSType` = `'pamc'`) maps 2-byte glyph indexes to 1-byte character codes, or to 1-byte glyph codes in bitmapped fonts whose font layouts do not exactly correspond to their script system's character encoding. The glyph-to-character table is conceptually the opposite of the TrueType character-code mapping table (type `'cmap'`). It is used by the WorldScript I contextual formatting routines.

Figure B-16 shows the format of the glyph-to-character table.

Figure B-16 Format of the glyph-to-character table



The table header has these elements:

- Version. The version number of this table. A value of $0100 means version 1.
- Format. The format code, a number that identifies the format of this table.
- Platform. The computer system this table is designed for. A value of 1 means Macintosh.
- Script code. The script system of this glyph set.
- Language code. The language of this glyph set.

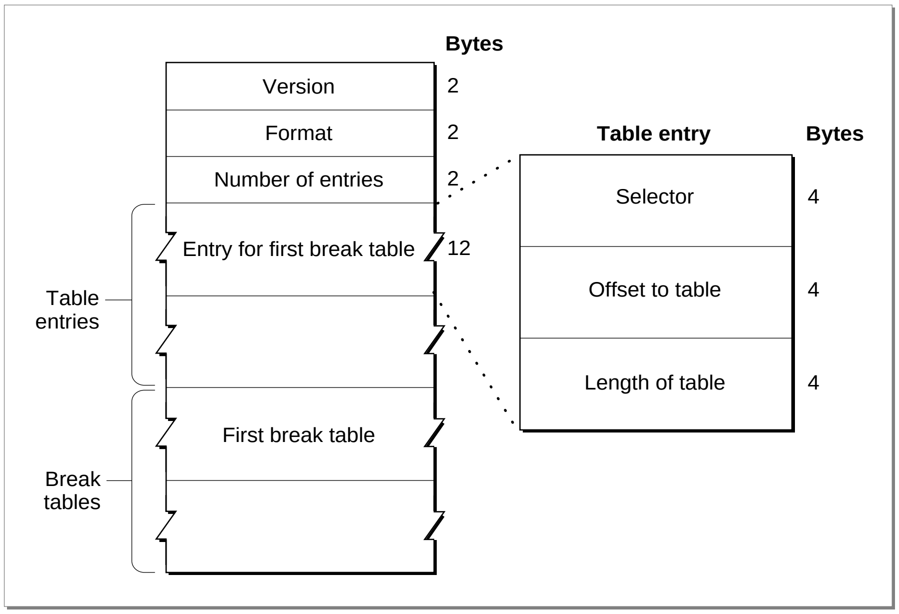
The table header is followed by an array of character code records. There is one record for each glyph index, which ranges from zero to a maximum value that can be greater than $FF. Each character code record has two elements:

- Character code. The character code corresponding to this glyph code.
- Script code. The script system of the character. For example, most glyphs that map to low-ASCII characters have a script code of `smRoman` in their character code record.

Break-Table Directory

The break-table directory (`OSType` = `'fwrd'`) provides access to one or more break tables (of type `NBreakTable`) for use by the Text Utilities `FindWordBreaks` procedure. It consists of a header, followed by entries that give offsets to the break tables, followed by the break tables themselves. Figure B-17 shows the format of the break-table directory.

Figure B-17 Format of the break-table directory



The directory header has these elements:

- Version. The version number of this directory. A value of $0100 means version 1.
- Format. Another type of version number.
- Number of entries. The number of entries, and therefore the number of break tables, in this directory.

The table entries consist of three elements each:

- Selector. A number that designates the specific type of break table referenced by this entry. The currently defined values are 0, signifying a table for word selection, and –1, signifying a table for line-breaking. These are the same default values that may be passed as break-table pointers to the `FindWordBreaks` procedure.
- Offset to table. The byte offset from the beginning of the directory to the break table referenced by this entry.
- Length of table. The length in bytes of the break table referenced by this entry.

The break tables themselves follow the table entries.

Most script systems' break tables are in their string-manipulation (`'itl2'`) resources. For some 1-byte complex script systems, break tables are in the encoding/rendering resource so that the Script Manager routines for replacing the WorldScript I script utilities will function correctly. See the discussions of the `GetScriptUtilityAddress` and `SetScriptUtilityAddress` routines in the chapter "Script Manager" in this book, and the discussion of WorldScript I in the appendix "Built-in Script Support."

Script Run Tables

Typically, tables to control the Text Utilities `FindScriptRun` function are in a script system's string-manipulation (`'itl2'`) resource. For some 1-byte complex script systems, the script run tables (`OSType` = `'fstb'`) are in the encoding/rendering resource so that the Script Manager routines for replacing the WorldScript I script utilities will function correctly.

The set of script run tables in the encoding/rendering resource consists of a header followed by one or more tables. The header has this format:

- Version number (2 bytes).
- Format code (2 bytes).
- Chain header (12 bytes). This part of the header is identical in format to the chain header in the line-layout metamorphosis table (see page B-63).

The header is followed by one or more tables. Each script run table consists of a table flags element (4 bytes), followed by a table identical to the new-format script run table in the string-manipulation resource. See "Script Run Table Format" beginning on page B-40.

For more information, see the discussions of the `GetScriptUtilityAddress` and `SetScriptUtilityAddress` routines in the chapter "Script Manager" in this book, and the discussion of WorldScript I in the appendix "Built-in Script Support."

Kashida Preferences Table

The kashida preferences table (`OSType` = `'kash'`), used in Arabic versions of the encoding/rendering resource, maps each glyph code to a kashida priority class. It specifies which glyphs can have kashida inserted between them, in what priority, when justifying Arabic text.

Feature List Table

The feature list table (`OSType` = `'flst'`) contains information used to override default line-layout behaviors (features) specified in the metamorphosis table (page B-63). It includes an array of feature entries, each of which specifies a feature type and a setting for that feature.

Reordering Table

The reordering table (`OSType` = `'reor'`) is a state table that specifies the classes and states used to reorder glyphs for contextual formatting. The reordering table contains offsets to three state tables and two arrays of level adjustments. The WorldScript I contextual formatting routine makes a first pass to resolve ordering of numbers, a second pass to resolve neutrals (whitespace, number separators, and terminators), and a third pass (using the values in the level adjustments arrays) to adjust nesting levels for each glyph. Finally, the routine reorders the line according to the resolved nesting levels.

Tables for 2-Byte Script Systems

In 2-byte script systems, the encoding/rendering resource contains byte-type and character-type information. The tables immediately follow the directory in the `'itl5'` header.

A byte-type table contains character-size information about a specific byte in the range of $00 to $FF. A character-type table contains detailed information about the character represented by a specific byte, given a particular character-encoding scheme.

Table B-9 shows the general structure of a typical encoding/rendering resource for a 2-byte script system.

Table B-9 Sample encoding/rendering resource for a 2-byte script system

Offset	Value	Explanation
0	$00010000	Version number (first release = 1.0)
4	2	Two tables in this resource
6	$000000000000	(reserved)
12	`'btyp'`	Tag for byte-type table
16	$00000000	(reserved)
20	30	Offset to the byte-type table

Table B-9 Sample encoding/rendering resource for a 2-byte script system (continued)

Offset	Value	Explanation
24	256	Length of the byte-type table
28	`'ctyp'`	Tag for the character-type table
32	$00000000	(reserved)
36	286	Offset to the character-type table
40	variable	Length of the character-type table
44		Start of byte-type table (256 bytes long)
300		Start of character-type table

Byte-Type Table

A byte-type table has 256 integer entries, one for each possible byte value in the range $00 to $FF. Each byte value is an index into the table. At each byte value, the table entry can have one of three values, specifying what kind of character or part of a character that byte value can represent.

- 1 = 1-byte character only
- 0 = 1-byte character or low-order byte of a 2-bye character
- –1 = high-order or low-order byte of a 2-byte character

When processing text sequentially in a buffer, you encounter a 2-byte character's high-order byte before its low-order byte. Thus you can determine the character relationship of a given byte (1-byte or 2-byte, high-order or low-order byte) by determining its byte type and, if necessary, comparing it with the byte type of the previous byte in the buffer.

Character-Type Table

The character-type table consists of one high-order byte table and a series of low-order byte tables.

The high-order byte table contains 256 word-length entries. The index position of each entry represents a high-order byte value. Nonzero entries mark valid high-order bytes of a 2-byte character. If a given entry is nonzero, it specifies which low-order byte table to consult to get character-type information.

There are one or more low-order byte tables, each of which can contain either a single entry or 256 word-length entries. If all low-order bytes for a given high-order byte have the same character type, the low-order byte "table" for that high-order byte consists of a single character-type value. Otherwise, every possible low-order byte is represented by an index into the table, with an appropriate character-type value at each valid index position.

For example, to find character-type information for the Japanese character with character code $EA40, you would examine location $EA in the high-order byte table; it would indicate the existence of a low-order byte table, and in that table you would examine location $40. That location would contain information showing that the character is a 2-byte JIS level-2 ideographic character that is part of the main character set.

Character types are discussed under the description of the `CharacterType` function in the chapter “Script Manager” in this book.

Transliteration Resource (Type 'trsl')

The transliteration resource (resource type `'trsl'`) contains information used by the Script Manager `TransliterateText` function, which performs phonetic conversion among subscripts in 2-byte script systems.

The transliteration resource is optional. Currently, no 1-byte script systems, including the Roman script system, have transliteration resources. All 2-byte script systems have transliteration resources.

The resource ID for a transliteration resource is within the range of resource ID numbers for its script system. There is one transliteration resource for each kind of transliteration supported by the script system. The name of an individual `'trsl'` resource, such as “Jamo to Hangul”, specifies the kind of transliteration that the resource performs.

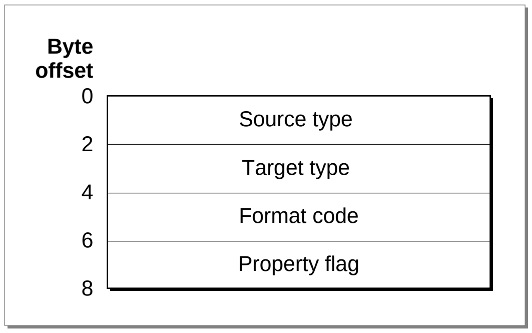
There are two formats for the transliteration resource: one supplies table-based transliteration from Jamo and Hangul, and vice versa, in the Korean script system; the other provides a more general rule-based transliteration.

Note
In the Roman script system, and for Roman text within other script systems, the `TransliterateText` function performs case conversion. The tables that control case conversion are in a script system’s string-manipulation (`'itl2'`) resource, not in a transliteration resource. ◆

Resource Header

Figure B-18 shows the format of the transliteration resource header.

Figure B-18 Format of the transliteration resource header



The resource header is the same for both the table-based and the rule-based formats:

- Source type. The type of text to perform the transliteration on. Specified by an integer; the currently defined mask constants for source type are listed under the discussion of the `TransliterateText` function in the chapter "Script Manager" in this book.
- Target type. The type of text to convert to. Specified by an integer; the currently defined target format constants are listed under the discussion of the `TransliterateText` function in the chapter "Script Manager" in this book.
- Format code. A number that identifies the format of this transliteration resource.
- Property flag. A bit field that specifies the kinds of operations to perform on a piece of text before or after transliteration. These are the currently defined bits of the property flag:

Bit number	Operation
1	Convert all 1-byte characters into 2-byte characters before performing the transliteration.
2	Convert all Roman characters to uppercase before performing the transliteration.

The property flag is needed because of the complex nature of the Chinese, Japanese, and Korean character sets, which include 1-byte and 2-byte characters as well as lowercase and uppercase characters. For example, to transliterate the Roman string "ki" into 2-byte Hiragana characters, the two-character string could be interpreted with as many as eight combinations of 1-byte Roman, 2-byte Roman, uppercase, and lowercase characters.

To simplify matters the transliteration resource allows you to convert your source text into a common set of characters before it matches them against the transliteration rules. So, to translate the Roman string "ki" into Hiragana, you can first convert the characters into their 2-byte equivalents, then convert them into uppercase, and then perform the transliteration.

Note
In most 2-byte transliteration resources, bits 1 and 2 in the property flag are set (= 1). The reason for the preliminary conversion of all source text to 2 bytes is that 2-byte Katakana is a superset of all the Katakana characters; thus, it is possible to convert all the 1-byte Katakana characters to 2-byte characters but not vice versa. ◆

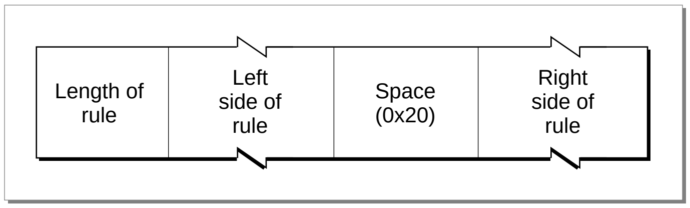
Rule-Based Format

In the rule-based version of the transliteration resource, the header is followed immediately by a 2-byte field containing a count of the number of rules in the resource; the rules immediately follow the count field and constitute the remainder of the resource. This is the definition of the rule-based resource header:

```
TYPE  RuleBasedTrslRecord =
   RECORD
      sourceType:    Integer; {target type for left side of rule}
      targetType:    Integer; {target type for right side of rule}
      formatNumber:  Integer; {format of this resource}
      propertyFlag:  Integer; {transliteration property flags}
      numberOfRules: Integer; {number of rules that follow}
   END;
```

Figure B-19 shows the format of a rule.

Figure B-19 Format of a transliteration rule



- The length of the rule is a byte that specifies the actual number of bytes in the rule, excluding the length byte itself.

- The left side of the rule contains the source pattern, a sequence of one or more character codes that the `TransliterateText` function compares to the source string. If it finds a match, it returns the right side of the rule (the target pattern). The rules are organized to implement the *longest match* algorithm, meaning that the longest source pattern that matches a particular target pattern is the one that is converted. For instance, the rule

  `abb` → `hello`

  takes precedence over the rule

  `ab` → `hello`

Some rules in some versions of the transliteration resource incorporate a look-ahead feature, in which a particular source pattern is converted to its target pattern only if it is followed by other specific characters. For example, if “[” represents the look-ahead symbol, the characters preceding it in the left side of the rule are converted to the right side of the rule only if the characters following “[” in the left side of the rule match the subsequent characters in the input string.

If these are the matching rules:

Left side	**Rignt side**
`a[bc`	`A`
`b`	`B`
`c`	`C`
`d[ef`	`D`
`f`	`F`

Then if we have the input string

```
abcdf
```

the output string will be:

```
ABCdF
```

because, in the input string, the characters “bc” follow “a”, but the characters “ef” do not follow “d”.

Table-Based Format

The Jamo-to-Hangul transliteration resource contains a set of conversion tables. The structure of the tables is private.

Jamo-to-Hangul transliteration is used by the input method supplied with the Korean script system; see the discussion of input methods in the chapter “Introduction to Text on the Macintosh” in this book.

Summary of the International Resources

Pascal Summary

Constants

```
{ Bits in the itlcFlags byte.}
itlcShowIcon = 7;                {show icon even if only one script}
itlcDualCaret = 6;               {use dual caret for mixed direction text}

{ Bits in the itlcSysFlags word.}
itlcSysDirection = 15;        {system direction - left/right or right/left}

{ the NumberParts indices }
tokLeftQuote = 1;
tokRightQuote = 2;
tokLeadPlacer = 3;
tokLeader = 4;
tokNonLeader = 5;
tokZeroLead = 6;
tokPercent = 7;
tokPlusSign = 8;
tokMinusSign = 9;
tokThousands = 10;
tokSeparator = 12;               {11 is a reserved field}
tokEscape = 13;
tokDecPoint = 14;
tokUnquoteds = 15;
tokMaxSymbols = 31;

curNumberPartsVersion = 1; {current version of NumberParts record}

currSymLead = 16;
currNegSym = 32;
currTrailingZ = 64;
currLeadingZ = 128;
```

```
zeroCycle = 1;                    {0:00 AM/PM format}
longDay = 0;                      {day of the month}
longWeek = 1;                     {day of the week}
longMonth = 2;                    {month of the year}
longYear = 3;                     {year}
supDay = 1;                       {suppress day of month}
supWeek = 2;                      {suppress day of week}
supMonth = 4;                     {suppress month}
supYear = 8;                      {suppress year}
dayLdingZ = 32;
mntLdingZ = 64;
century = 128;
secLeadingZ = 32;
minLeadingZ = 64;
hrLeadingZ = 128;

{ Date Orders }
mdy = 0;
dmy = 1;
ymd = 2;
myd = 3;
dym = 4;
ydm = 5;
```

Data Types

```
TYPE  ItlcRecord =
      RECORD
         itlcSystem:        Integer;    {default system script}
         itlcReserved:      Integer;    {reserved}
         itlcFontForce:     SignedByte; {default font force flag}
         itlcIntlForce:     SignedByte; {default intl force flag}
         itlcOldKybd:       SignedByte; {MacPlus intl keybd flag}
         itlcFlags:         SignedByte; {general flags}
         itlcIconOffset:    Integer;    {reserved}
         itlcIconSide:      SignedByte; {reserved}
         itlcIconRsvd:      SignedByte; {reserved}
         itlcRegionCode:    Integer;    {preferred verXxx code}
         itlcSysFlags:      Integer;    {flags for setting system globals}
         itlcReserved4:     ARRAY[0..31] OF SignedByte;      {for future use}
      END;
```

```
ItlbRecord =
RECORD
   itlbNumber:    Integer;       {itl0 id number}
   itlbDate:      Integer;       {itl1 id number}
   itlbSort:      Integer;       {itl2 id number}
   itlbFlags:     Integer;       {Script flags}
   itlbToken:     Integer;       {itl4 id number}
   itlbEncoding:  Integer;       {itl5 ID # (optional; char encoding)}
   itlbLang:      Integer;       {current language for script }
   itlbNumRep:    SignedByte;    {number representation code}
   itlbDateRep:   SignedByte;    {date representation code }
   itlbKeys:      Integer;       {KCHR id number}
   itlbIcon:      Integer;       {ID# of SICN or kcs#/kcs4/kcs8 family}
END;

ItlbExtRecord =
RECORD
   base:             ItlbRecord;      {unextended ItlbRecord}
   itlbLocalSize:    LongInt;         {size of script's local record}
   itlbMonoFond:     Integer;         {default monospace FOND ID}
   itlbMonoSize:     Integer;         {default monospace font size}
   itlbPrefFond:     Integer;         {preferred FOND ID}
   itlbPrefSize:     Integer;         {preferred font size}
   itlbSmallFond:    Integer;         {default small FOND ID}
   itlbSmallSize:    Integer;         {default small font size}
   itlbSysFond:      Integer;         {default system FOND ID}
   itlbSysSize:      Integer;         {default system font size}
   itlbAppFond:      Integer;         {default application FOND ID}
   itlbAppSize:      Integer;         {default application font size}
   itlbHelpFond:     Integer;         {default Help Mgr FOND ID}
   itlbHelpSize:     Integer;         {default Help Mgr font size}
   itlbValidStyles:  Style;           {set of valid styles for script}
   itlbAliasStyle:   Style;           {style (set) to mark aliases}
END;

Intl0Rec =
PACKED RECORD
   decimalPt:     Char; {decimal point character}
   thousSep:      Char; {thousands separator character}
   listSep:       Char; {list separator character}
   currSym1:      Char; {currency symbol}
   currSym2:      Char;
   currSym3:      Char;
   currFmt:       Byte; {currency format flags}
   dateOrder:     Byte; {order of short date elements: mdy, dmy, etc.}
   shrtDateFmt:   Byte; {format flags for each short date element}
   dateSep:       Char; {date separator character}
   timeCycle:     Byte; {specifies time cycle: 0..23, 1..12, or 0..11}
   timeFmt:       Byte; {format flags for each time element}
   mornStr:       PACKED ARRAY[1..4] OF Char;
                        {trailing string for AM if 12-hour cycle}
   eveStr:        PACKED ARRAY[1..4] OF Char;
                        {trailing string for PM if 12-hour cycle}
   timeSep:       Char; {time separator character}
   time1Suff:     Char; {trailing string for AM if 24-hour cycle}
   time2Suff:     Char;
   time3Suff:     Char;
   time4Suff:     Char;
   time5Suff:     Char; {trailing string for PM if 24-hour cycle}
   time6Suff:     Char;
   time7Suff:     Char;
   time8Suff:     Char;
   metricSys:     Byte; {255 if metric, 0 if inches etc.}
   intl0Vers:     Integer;
                        {region code (hi byte) and version (lo byte)}
END;
Intl0Ptr = ^Intl0Rec;
Intl0Hndl = ^Intl0Ptr;

Intl1Rec =
PACKED RECORD
   days:          ARRAY[1..7] OF Str15;      {day names}
   months:        ARRAY[1..12] OF Str15;     {month names}
   suppressDay:   Byte;
                     {255 for no day, or flags to suppress any element}
   lngDateFmt:    Byte;       {order of long date elements}
   dayLeading0:   Byte;       {255 for leading 0 in day number}
   abbrLen:       Byte;       {length for abbreviating names}
   st0:           PACKED ARRAY[1..4] OF Char;
                              {separator strings for long date format}
   st1:           PACKED ARRAY[1..4] OF Char;
   st2:           PACKED ARRAY[1..4] OF Char;
   st3:           PACKED ARRAY[1..4] OF Char;
   st4:           PACKED ARRAY[1..4] OF Char;
   intl1Vers:     Integer;
                        {region code (hi byte) and version (lo byte)}
   localRtn:      ARRAY[0..0] OF Integer;
                        {a flag for optional extension}
```

```
      END;
      Intl1Ptr = ^Intl1Rec;
      Intl1Hndl = ^Intl1Ptr;

      Itl1ExtRec =
      RECORD
         base:                   Intl1Rec;{un-extended Intl1Rec}
         version:                Integer; {version number}
         format:                 Integer; {format code}
         calendarCode:           Integer; {calendar code for 'itl1'}
         extraDaysTableOffset:   LongInt; {offset to extra days table}
         extraDaysTableLength:   LongInt; {length of extra days table}
         extraMonthsTableOffset: LongInt; {offset to extra months table}
         extraMonthsTableLength: LongInt; {length of extra months table}
         abbrevDaysTableOffset:  LongInt; {offset to abbrev. days table}
         abbrevDaysTableLength:  LongInt; {length of abbrev. days table}
         abbrevMonthsTableOffset:LongInt; {offset to abbr. months table}
         abbrevMonthsTableLength:LongInt; {length of abbr. months table}
         extraSepsTableOffset:   LongInt; {offset to extra seps table}
         extraSepsTableLength:   LongInt; {length of extra seps table}
         tables:                 ARRAY[0..0] OF Integer;
                                    {the tables; variable-length}
END;

      NItl4Rec =
      RECORD
         flags:            Integer; {reserved}
         resourceType:     LongInt; {contains 'itl4'}
         resourceNum:      Integer; {resource ID}
         version:          Integer; {version number}
         format:           Integer; {format code}
         resHeader:        Integer; {reserved}
         resHeader2:       LongInt; {reserved}
         numTables:        Integer; {number of tables, one-based}
         mapOffset:        LongInt; {table that maps byte to token}
         strOffset:        LongInt; {routine that copies string}
         fetchOffset:      LongInt; {routine to get next byte of character}
         unTokenOffset:    LongInt; {table that maps token to string}
         defPartsOffset:   LongInt; {offset to default number parts table}
         whtSpListOffset:  LongInt; {offset to whitespace table}
         resOffset7:       LongInt; {reserved}
         resOffset8:       LongInt; {reserved}
         resLength1:       Integer; {reserved}
         resLength2:       Integer; {reserved}
         resLength3:       Integer; {reserved}
         unTokenLength:    Integer; {length of untoken table}
         defPartsLength:   Integer; {length of number parts table}
         whtSpListLength:  Integer; {length of whitespace table}
         resLength7:       Integer; {reserved}
         resLength8:       Integer; {reserved}
      END;
      NItl4Ptr = ^NItl4Rec;
      NItl4Handle = ^NItl4Ptr;

      UntokenTable =
      RECORD
         len:        Integer;
         lastToken:  Integer;
         index:      ARRAY[0..255] OF Integer;  {index table; last=lastToken}
      END;
      UntokenTablePtr = ^UntokenTable;
      UntokenTableHandle = ^UntokenTablePtr;

WideChar = RECORD
    CASE Boolean OF
      TRUE:
        (a: PACKED ARRAY[0..1] OF Char);{0 is the high order character}
      FALSE:
        (b: Integer);
    END;

WideCharArr = RECORD
    size: Integer;
    data: PACKED ARRAY[0..9] OF WideChar;
    END;

      NumberParts =
      RECORD
         version:       Integer;
         data:          ARRAY[1..31] OF WideChar;
         pePlus:        WideCharArr;
         peMinus:       WideCharArr;
         peMinusPlus:   WideCharArr;
         altNumTable:   WideCharArr;
         reserved:      PACKED ARRAY[0..19] OF Char;
      END;
      NumberPartsPtr = ^NumberParts;
```

```
Itl5Record =
RECORD
   versionNumber:    Fixed;         {itl5 resource version number}
   numberOfTables:   Integer;       {number of tables it contains}
   reserved:         ARRAY[0..2] OF Integer;
                                    {reserved for internal use}
   tableDirectory:   ARRAY[0..0] OF TableDirectoryRecord;
                                    {table directory records}
END;

TableDirectoryRecord =
RECORD
   tableSignature:   OSType ;    {4 byte long table name}
   reserved:         LongInt;    {reserved for internal use}
   tableStartOffset: LongInt ;   {table start offset in bytes}
   tableSize:        LongInt;    {table size in bytes}
END;

RuleBasedTrslRecord =
RECORD
   sourceType:    Integer;    {target type for left side of rule}
   targetType:    Integer;    {target type for right side of rule}
   formatNumber:  Integer;    {transliteration resource format number}
   propertyFlag:  Integer;    {transliteration property flags}
   numberOfRules: Integer;    {Number of rules following this field}
END;
```

C Summary

Constants

```
enum {

/* Bits in the itlcFlags byte. */
 itlcShowIcon = 7,            /*show icon even if only one script*/
 itlcDualCaret = 6,           /*use dual caret for mixed direction text*/

/* Bits in the itlcSysFlags word. */
 itlcSysDirection = 15,       /*System direction--left/right or right/left*/
```

```
/* the NumberParts indices */
 tokLeftQuote = 1,
 tokRightQuote = 2,
 tokLeadPlacer = 3,
 tokLeader = 4,
 tokNonLeader = 5,

tokZeroLead = 6,
 tokPercent = 7,
 tokPlusSign = 8,
 tokMinusSign = 9,
 tokThousands = 10,
 tokSeparator = 12,             /*11 is a reserved field*/
 tokEscape = 13,
 tokDecPoint = 14,
 tokUnquoteds = 15,
 tokMaxSymbols = 31,

   curNumberPartsVersion = 1        /*current version of NumberParts record*/
};

enum {
 currSymLead = 16,
 currNegSym = 32,
 currTrailingZ = 64,
 currLeadingZ = 128,
};

enum {mdy,dmy,ymd,myd,dym,ydm};

enum {
 zeroCycle = 1,                 /*0:00 AM/PM format*/
 longDay = 0,                   /*day of the month*/
 longWeek = 1,                  /*day of the week*/
 longMonth = 2,                 /*month of the year*/
 longYear = 3,                  /*year*/
 supDay = 1,                    /*suppress day of month*/
 supWeek = 2,                   /*suppress day of week*/
 supMonth = 4,                  /*suppress month*/
 supYear = 8,                   /*suppress year*/
 dayLdingZ = 32,
 mntLdingZ = 64,
 century = 128,
 secLeadingZ = 32,
 minLeadingZ = 64,
 hrLeadingZ = 128
};
```

Data Types

```
typedef unsigned char DateOrders;

struct ItlcRecord {
 short itlcSystem;         /*default system script*/
 short itlcReserved;       /*reserved*/
 char itlcFontForce;       /*default font force flag*/
 char itlcIntlForce;       /*default intl force flag*/
 char itlcOldKybd;         /*MacPlus intl keybd flag*/
 char itlcFlags;           /*general flags*/
 short itlcIconOffset;     /*reserved*/
 char itlcIconSide;        /*reserved*/
 char itlcIconRsvd;        /*reserved*/
 short itlcRegionCode;     /*preferred verXxx code*/
 short itlcSysFlags;       /*flags for setting system globals*/
 char itlcReserved4[32];   /*for future use*/
};
typedef struct ItlcRecord ItlcRecord;

struct ItlbRecord {
 short itlbNumber;         /*itl0 id number*/
 short itlbDate;           /*itl1 id number*/
 short itlbSort;           /*itl2 id number*/
 short itlbFlags;          /*Script flags*/
 short itlbToken;          /*itl4 id number*/
 short itlbEncoding;       /*itl5 ID # (optional; char encoding)*/
 short itlbLang;           /*current language for script */
 char itlbNumRep;          /*number representation code*/
 char itlbDateRep;         /*date representation code */
 short itlbKeys;           /*KCHR id number*/
 short itlbIcon;           /*ID # of SICN or kcs#/kcs4/kcs8 family.*/
};
typedef struct ItlbRecord ItlbRecord;

/* New ItlbExtRecord structure for System 7 */
struct ItlbExtRecord {
 ItlbRecord base;          /*unextended ItlbRecord*/
 long itlbLocalSize;       /*size of script's local record*/
 short itlbMonoFond;       /*default monospace FOND ID*/
 short itlbMonoSize;       /*default monospace font size*/
 short itlbPrefFond;       /*preferred FOND ID*/

short itlbPrefSize;        /*preferred font size*/
 short itlbSmallFond;      /*default small FOND ID*/
 short itlbSmallSize;      /*default small font size*/
 short itlbSysFond;        /*default system FOND ID*/
 short itlbSysSize;        /*default system font size*/
 short itlbAppFond;        /*default application FOND ID*/
 short itlbAppSize;        /*default application font size*/
 short itlbHelpFond;       /*default Help Mgr FOND ID*/
 short itlbHelpSize;       /*default Help Mgr font size*/
 Style itlbValidStyles;    /*set of valid styles for script*/
 Style itlbAliasStyle;     /*style (set) to mark aliases*/
};
typedef struct ItlbExtRecord ItlbExtRecord;

struct Intl0Rec {
 char decimalPt;/*decimal point character*/
 char thousSep;/*thousands separator character*/
 char listSep; /*list separator character*/
 char currSym1;/*currency symbol*/
 char currSym2;
 char currSym3;
 unsigned char currFmt;       /*currency format flags*/
 unsigned char dateOrder;     /*order of short date elements:mdy,dmy,etc.*/
 unsigned char shrtDateFmt;   /*format flags for each short date element*/
 char dateSep;                /*date separator character*/
 unsigned char timeCycle;     /*specifies time cycle:0..23,1..12,or 0..11*/
 unsigned char timeFmt;       /*format flags for each time element*/
 char mornStr[4];             /*trailing string for AM if 12-hour cycle*/
 char eveStr[4];              /*trailing string for PM if 12-hour cycle*/
 char timeSep;                /*time separator character*/
 char time1Suff;              /*trailing string for AM if 24-hour cycle*/
 char time2Suff;
 char time3Suff;
 char time4Suff;
 char time5Suff;              /*trailing string for PM if 24-hour cycle*/
 char time6Suff;
 char time7Suff;
 char time8Suff;
 unsigned char metricSys;     /*255 if metric, 0 if inches etc.*/
 short intl0Vers;             /*region code (hi byte) and version (lo byte)*/
```

```
};
typedef struct Intl0Rec Intl0Rec;
typedef Intl0Rec *Intl0Ptr, **Intl0Hndl;

struct Intl1Rec {
 Str15 days[7];               /*day names*/
 Str15 months[12];            /*month names*/
 unsigned char suppressDay;   /*255 = no day, or flags to suppress elements*/
 unsigned char lngDateFmt;    /*order of long date elements*/
 unsigned char dayLeading0;   /*255 for leading 0 in day number*/
 unsigned char abbrLen;       /*length for abbreviating names*/
 char st0[4];                 /*separator strings for long date format*/
 char st1[4];
 char st2[4];
 char st3[4];
 char st4[4];
 short intl1Vers;             /*region code (hi byte) and version (lo byte)*/
 short localRtn[1];           /*now a flag for opt extension*/
};
typedef struct Intl1Rec Intl1Rec;
typedef Intl1Rec *Intl1Ptr, **Intl1Hndl;

struct Itl1ExtRec {              /*fields for optional itl1 extension*/
 Intl1Rec base;                  /*un-extended Intl1Rec*/
 short version;
 short format;
 short calendarCode;             /*calendar code for this itl1 resource*/
 long extraDaysTableOffset;      /*offset in itl1 to extra days table*/
 long extraDaysTableLength;      /*length of extra days table*/
 long extraMonthsTableOffset;    /*offset in itl1 to extra months table*/
 long extraMonthsTableLength;    /*length of extra months table*/
 long abbrevDaysTableOffset;     /*offset in itl1 to abbrev days table*/
 long abbrevDaysTableLength;     /*length of abbrev days table*/
 long abbrevMonthsTableOffset;   /*offset in itl1 to abbrev months table*/
 long abbrevMonthsTableLength;   /*length of abbrev months table*/
 long extraSepsTableOffset;      /*offset in itl1 to extra seps table*/
 long extraSepsTableLength;      /*length of extra seps table*/
 short tables[1];                /*now a flag for opt extension*/
};
typedef struct Itl1ExtRec Itl1ExtRec;

struct UntokenTable {
 short len;
 short lastToken;
 short index[256];            /*index table; last = lastToken*/
};
typedef struct UntokenTable UntokenTable;
typedef UntokenTable *UntokenTablePtr, **UntokenTableHandle;

union WideChar {
 char a[2];                /*0 is the high-order character*/
 short b;
};
typedef union WideChar WideChar;

struct WideCharArr {
 short size;
 WideChar data[10];
};
typedef struct WideCharArr WideCharArr;

struct NumberParts {
 short version;
 WideChar data[31];        /*index by [tokLeftQuote..tokMaxSymbols]*/
 WideCharArr pePlus;
 WideCharArr peMinus;
 WideCharArr peMinusPlus;
 WideCharArr altNumTable;
 char reserved[20];
};
typedef struct NumberParts NumberParts;
typedef NumberParts *NumberPartsPtr;

/* New NItl4Rec for System 7.0: */
struct NItl4Rec {
 short flags;           /*reserved*/
 long resourceType;     /*contains 'itl4'*/
 short resourceNum;     /*resource ID*/
 short version;         /*version number*/
 short format;          /*format code*/
 short resHeader;       /*reserved*/
 long resHeader2;       /*reserved*/
 short numTables;       /*number of tables, one-based*/
 long mapOffset;        /*offset to table that maps byte to token*/
 long strOffset;        /*offset to routine that copies canonical string*/
 long fetchOffset;      /*offset to routine that gets next byte of char.*/
 long unTokenOffset;    /*offset to table that maps token to canon. string*/
 long defPartsOffset;   /*offset to number parts table*/
 long whtSpListOffset;  /*offset to whitespace table*/
 long resOffset7;       /*reserved*/
 long resOffset8;       /*reserved*/
 short resLength1;      /*reserved*/
 short resLength2;      /*reserved*/
 short resLength3;      /*reserved*/
 short unTokenLength;   /*length of untoken table*/
 short defPartsLength;  /*length of default number parts table*/
 short whtSpListLength; /*length of whitespace table*/
 short resLength7;      /*reserved*/
 short resLength8;      /*reserved*/
};
typedef struct NItl4Rec NItl4Rec;
typedef NItl4Rec *NItl4Ptr, **NItl4Handle;

struct TableDirectoryRecord {
 OSType tableSignature;          /*4 byte long table name */
 unsigned long reserved;         /*reserved for internal use */
 unsigned long tableStartOffset; /*table start offset in byte*/
 unsigned long tableSize;        /*table size in byte*/
 };
typedef struct TableDirectoryRecord TableDirectoryRecord;

struct Itl5Record {
 Fixed versionNumber;                     /*itl5 resource version number */
 unsigned short numberOfTables;           /*number of tables it contains */
 unsigned short reserved[3];              /*reserved for internal use */
 TableDirectoryRecord tableDirectory[1];  /*table directory records */
 };
typedef struct Itl5Record Itl5Record;

struct RuleBasedTrslRecord {
 short sourceType;      /*target type for left side of rule */
 short targetType;      /*target type for right side of rule */
 short formatNumber;    /*transliteration resource format number */
 short propertyFlag;    /*transliteration property flags */
 short numberOfRules;   /*number of rules following this field */
 };
typedef struct RuleBasedTrslRecord RuleBasedTrslRecord;
```