Text Utilities

The Text Utilities provide you with an integrated collection of routines for performing a variety of operations on textual information, ranging from modifying the contents of a string, to sorting strings from different languages, to converting times, dates, and numbers from internal representations to formatted strings and back. These routines work in conjunction with QuickDraw text drawing routines to help you display and modify text in applications that are distributed to an international audience.

Many of the Text Utilities routines were previously located in other managers in the Macintosh system software. Several of these have been replaced with new versions that take a script code as a parameter and others have been renamed. The appendix "Renamed and Relocated Text Routines" in this book shows the original names and locations of all of the text-handling routines.

You need to read this chapter if you are working with text in your application. This includes basic operations such as accessing a string resource and comparing two strings for equality. If you have used Macintosh text-processing routines in the past, you need to review the material in this chapter to understand the new capabilities that have been added to many of the routines.

To understand the material in this chapter, you need to have a basic understanding of the Macintosh script management system. Read this chapter after reading "Introduction to Text on the Macintosh." For parts that describe international resources, read the appendix "International Resources" along with this chapter. For parts that describe text layout, read "QuickDraw Text" along with this chapter.

This chapter describes the resources and text strings with which the Text Utilities interact, and discusses how to use the Text Utilities to compare, sort, modify, and find breaks in text strings, and to convert and format date, time, and numeric strings.

About the Text Utilities

The Text Utilities routines are used for numerous text-handling tasks, including

- defining strings—including functions for allocating strings in the heap and for loading strings from resources
- comparing and sorting strings—including functions for testing whether two strings are equal and functions for finding the sorting relationship between two strings
- modifying the contents of strings—including routines for converting the case of characters, stripping diacritical marks, replacing substrings, and truncating strings
- finding breaks and boundaries in text—including routines for finding word and line breaks, and for finding different script runs in a line of text

- converting and formatting date and time strings—including routines that convert numeric and string representations of dates and times into record format, and routines that convert numeric and record representations of dates and times into strings
- converting and formatting numeric strings—including routines that convert string representations of numbers into numeric representations, and routines that convert from numeric representations into formatted strings

The Text Utilities and the International Resources

Many of the Text Utilities routines script-aware, which means that you need to understand script systems and the international resources to use the routines properly. Each script system contains a collection of these resources, which contain data and routines that define how regional differences are handled. In particular, the international resources contain tables that define how different text elements are represented.

The resources used by the Text Utilities are

- the script-sorting (`'itlm'`) resource, which defines the sorting order among scripts
- the numeric-format (`'itl0'`) resource, which describes details of how time, numbers, and short dates are presented
- the long-date-format (`'itl1'`) resource, which describes how long dates are presented
- the string-manipulation (`'itl2'`) resource, which contains tables that define how strings are sorted and which characters cause breaks between words
- the tokens (`'itl4'`) resource, which contains tables that define which sequences of characters create which tokens

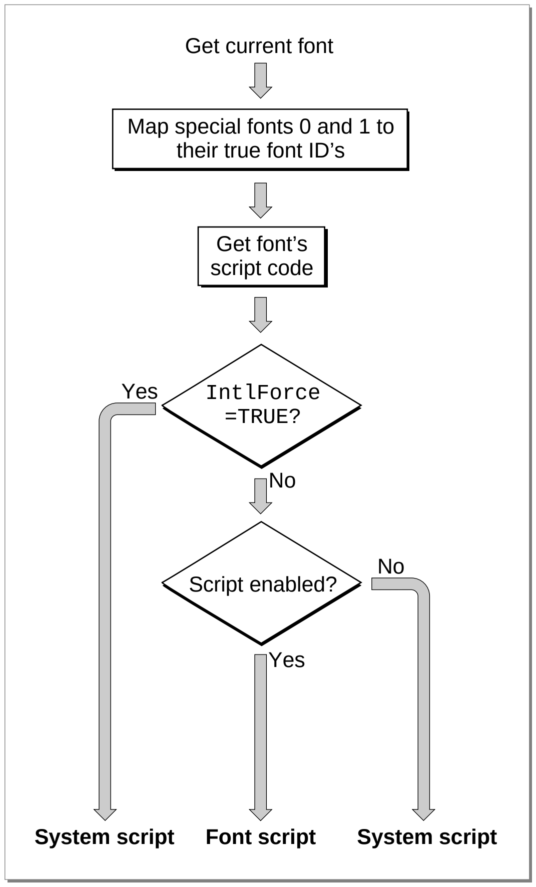
The appendix "International Resources" in this book describes each international resource in detail.

Obtaining Resource Information

Many Text Utilities routines perform operations—such as modifying text or sorting strings—that require information from resources. Some routines determine which resources to use by checking a resource parameter; others check a script code parameter.

For a Text Utilities routine that uses a resource parameter, you can explicitly specify the resource you want to use, or you can specify `NIL`. The value `NIL` causes the routine to use the resources associated with the current script. The **current script** is either the system script (the script associated with the currently running version of Macintosh system software) or the font script (the script of the current font in the current graphics port), and is determined by the value of the international resources selection flag, which is represented by the global variable `IntlForce`. If the value of this flag is `TRUE`, the current script is the system script; if the value of the flag is `FALSE`, the current script is the font script. (See Figure 5-1.)

Figure 5-1 Determining the current script



The international resources selection flag is initialized at startup from the system script configuration (`'itlc'`) resource. For most system scripts, the international resources selection flag has a default value of `TRUE`. If you want to change its value, you can use the `SetSMVariable` function with the `smIntlForce` selector. If you want to test its values, you can use the `GetSMVariable` function with the same selector. The `GetSMVariable` and `SetSMVariable` functions are described in the chapter "Script Manager" in this book.

The value of the international resources selection flag actually controls the operation of the `GetIntlResource` function, which is used by other routines to access the international resources. The operation of `GetIntlResource` is described in detail in the chapter "Script Manager" in this book.

Other Text Utilities routines use a script code parameter, in which you specify the unique number that defines the script system whose resources you want to use.

Constants for all defined script codes are listed in the chapter "Script Manager" in this book. If you wish, you can specify the following two special constants in the script code parameter: `smSystemScript`, which indicates that the routine should use the international resources of the system script, and `smCurrentScript`, which indicates that the routine should use the font script.

Pascal Strings and Text Strings

This chapter describes many routines, almost all of which operate on strings that are specified in one of two forms: as Pascal strings or as text strings. These are two ways of representing text characters, each of which has advantages and disadvantages relative to the other.

A **Pascal string** is an array of characters, the first byte of which defines the number of bytes that follow. This is the standard representation of strings used in Pascal programming. Most of the Text Utilities routines that use Pascal strings use the `Str255` or `StringHandle` type. An advantage of the `Str255` type is that it can be passed directly as a single parameter on the stack. A disadvantage is that a `Str255` value can hold only up to 255 bytes of character data. A typical Pascal string parameter declaration is as follows:

```
PROCEDURE MyUsePascalString (str: Str255);
```

The alternative representation for character data, a **text string**, can contain up to 32,767 bytes of character data and is specified by two parameters: a pointer to the first byte of character data and a 16-bit integer length value. A typical declaration of a routine that uses a text string parameter declaration is as follows:

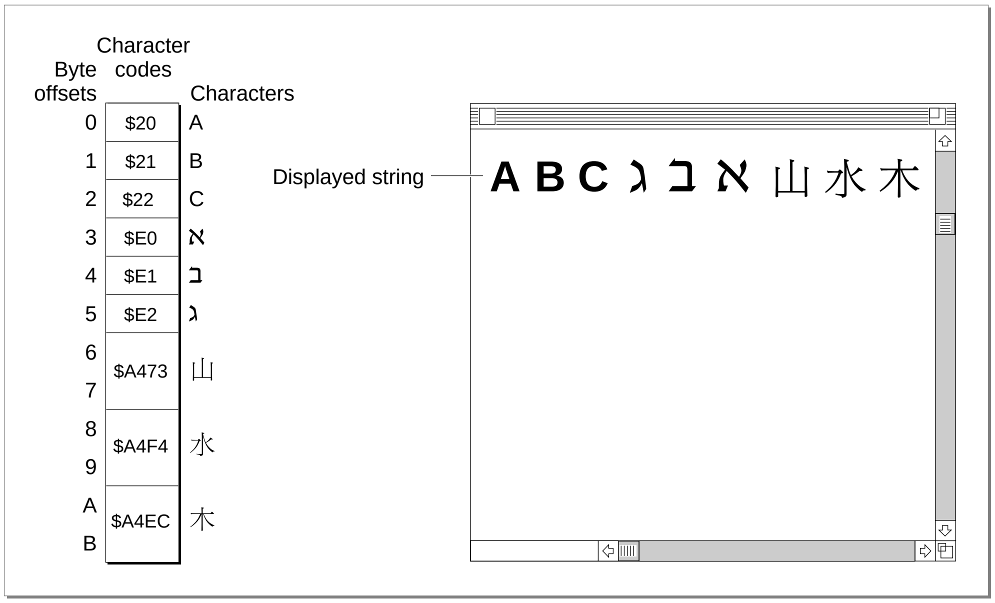
```
PROCEDURE MyUseTextString (textPtr: Ptr; textLen: Integer);
```

Some of the Text Utilities routines have been modified to allow for even longer strings. These routines allow a 32-bit integer length value, which means that they can operate on text strings of up to approximately two billion bytes in length.

IMPORTANT

Length is specified in bytes, not characters, for both Pascal strings and text strings. In international text processing, two bytes are sometimes required to represent a single character in certain fonts, as illustrated in Figure 5-2. Because you have to accommodate both 1-byte and 2-byte characters in the same string, the length of each string cannot be specified as a number of characters. ▲

Figure 5-2 A string containing 1-byte and 2-byte characters



Using the Text Utilities

This section provides you with general information about how to use each group of Text Utilities routines. A description of the basic concepts that each group deals with is provided so that you can learn how to choose the appropriate function to meet your needs. The following areas are covered:

- how to define strings
- how to retrieve the values of string resources
- how to compare and sort strings, including how to sort strings in a multi-language environment
- how to modify strings by converting between uppercase and lowercase and stripping diacritical marks
- how to truncate strings to fit in specified screen areas
- how to search for and replace portions of strings
- how to find word, line, and script run boundaries in strings

- how to convert date and time values into strings, accommodating different international formats for the strings
- how to convert date and time strings into internal numeric representations
- how to convert numeric values into strings, accommodating different international formats for the numbers

Defining Strings

Before you use a string in your application, you must define it in some way. You can use a string variable to represent text characters, or you can allocate an object in the heap to represent those text characters. Many strings, including error messages and lists of input choices, originate in the resource fork of your application; you need access to these string resources before using any of the Text Utilities routines with the strings. This section describes several routines that you can use to allocate strings and to access string resources.

Working With String Handles

When you need to modify a string, it is useful to create a version of the string as an object in the heap. That way, you can increase or reduce the memory allocated for the string object as necessary. String handles provide a means for you to work with heap-oriented strings.

The Text Utilities include two routines that work with strings and string handles. The `NewString` function creates a copy of a specified string in the heap and returns a handle to that string. The `SetString` procedure changes the contents of the string referenced by a string handle by copying a specified string into its area of the heap. If you pass `SetString` a string that is longer than the one originally referenced by the handle, `SetString` automatically resizes the handle and copies in the contents of the specified string.

For example, suppose that you want to write a routine that creates both an uppercase and lowercase version of an input string. The `MyUpperAndLower` function shown in Listing 5-1 replaces the contents of the `lowerStr` parameter with a lowercase version of the `str` parameter, and returns a new string handle to the uppercase version.

Listing 5-1 Using the `NewString` and `SetString` routines

```
FUNCTION MyUpperAndLower (str: Str255;
                           VAR lowerStr: StringHandle): StringHandle;
VAR
   myStr: StringHandle;
   len: Integer;
BEGIN
   myStr := NewString(str);   {create string handle to be converted to upper}
   SetString( lowerStr, str); {set string handle to be converted to lower}
```

```
    len := ord(str[0]);

    HLock(Handle(myStr));      {UppercaseText can move memory}
    UppercaseText(@myStr^^[1], len, smSystemScript);
    HLock(Handle(myStr));

    HLock(Handle(lowerStr));      {LowercaseText can move memory}
    LowercaseText(@lowerStr^^[1], len, smSystemScript);
    HLock(Handle(lowerStr));
    MyUpperAndLower := myStr;
END;
```

Working With String Resources

Since many of the strings in your Macintosh applications are specified in resource files, you need access to those strings. Strings are defined by two different resource types: the string (`'STR '`) resource and the string list (`'STR#'`) resource. To work with the string resource, you use the `GetString` function, and to work with a string list resource, you use the `GetIndString` procedure.

The `GetString` function reads a string resource into memory and returns the handle to the string resource as its result. `GetString` does not copy the string, so you must create your own copy if you are going to modify the string in your application. If the resource has already been read into memory, `GetString` simply returns a handle to the string.

If you use a number of strings in your application, it is more efficient to specify them in a string list resource rather than as individual resources. This is because the system software that reads in the resources can operate more efficiently when reading a collection of strings from a file than when reading and storing each individually.

To work with an element in a string list, use the `GetIndString` procedure. It reads the resource, locates the string, and copies the string into a Pascal string variable you supply. You can then use the `NewString` function to create a copy of the string in the heap, if you wish.

Sorting Strings in Different Languages

Strings in the same language must be sorted according to that language's sorting rules; information about these rules is found in resources that belong to that language's script system.

However, if the strings are in two different languages or writing systems, sorting order is also governed by rules about the order among languages or writing systems. For example, an application might need to sort names in English, French, and German, or it might need to sort an index of English and Japanese names written in Roman and Katakana characters.

If you only need to sort strings from a single language in your application, you don't need to read this section or use the routines that are described here. You can skip ahead to the section "Sorting Strings in the Same Language," which begins on page 5-12.

When you sort strings in different languages, you must use routines that work with script-sorting and language-sorting information. The script-sorting (`'itlm'`) resource contains tables that define the sorting order among the languages or writing systems in each script system and among the different script systems that are available. It also shows the parent script for each language, the parent language for each region, and the default language for each script. The `ScriptOrder` function, which determines the sorting relationship between two script systems, and the `LanguageOrder` function, which determines the sorting relationship between two languages, use the tables in the script-sorting resource to determine their results. For more information on the script-sorting resource, see the appendix "International Resources" in this book.

To sort two Pascal strings in different languages, you begin by calling the `StringOrder` function. (You can use the `TextOrder` function to compare two text strings; it operates in the same way as does `StringOrder`.) The `StringOrder` function first calls the `ScriptOrder` function; if the script codes of the two strings are different, then `StringOrder` returns a result indicating the sorting relationship between the two script codes. For example, if the first string is from a Japanese script system and the second is from a Thai script system, then the second string comes before the first according to the tables in the script-sorting resource.

If both strings come from the same script system, `StringOrder` then compares their language codes by calling the `LanguageOrder` function. If the language codes of the two strings are different, then `StringOrder` returns a result indicating the sorting relationship between the two language codes. For example, if the first string is in English and the second is in German, then the first string comes before the second according to the tables in the script-sorting resource.

Finally, if the script codes and language codes for both strings are the same, then `StringOrder` compares the two strings using one of the comparison functions described in the next section, "Sorting Strings in the Same Language."

If you need to sort a collection of strings, you can choose to implement your sorting algorithm so that it uses `StringOrder` or `TextOrder`, or you can build a list for each language and/or for each script system and sort each list independently. If you want to use `StringOrder` or `TextOrder`, you need to store each string so that you can easily access its script code and language code during the sort.

It is usually desirable to sort all strings from a script system together, using the sorting rules that are associated with the current language for that script on the machine (and ignoring the different sorting rules for the different languages). For example, if you are sorting German, French, and English strings together for a system in England, you usually want the English sorting rules to be applied to all of those strings. In some cases, it may be more efficient to build a list for each language by using the language code of each to determine to which list it belongs. After building a list for each language, you can sort each with an algorithm that uses one of the comparison functions described in the next section, "Sorting Strings in the Same Language."

Figure 5-3 shows a collection of strings from different languages that need to be sorted.

Figure 5-3 Strings in different languages in one list

Script	Language	String
Roman	English	love
	German	Liebe
Japanese	Japanese	愛
Roman	English	peace
	German	Frieden
Japanese	Japanese	平和
Roman	English	hope
	German	Hoffnung
Japanese	Japanese	希望

In Figure 5-4, the strings have been sorted into two lists: one for each script system. The Roman script system strings have been sorted according to the sorting rules for English, which is assumed to be the current language for the script in this example.

Figure 5-4 Strings in different languages sorted by script

Script	Strings
Roman	Frieden Hoffnung hope Liebe love peace
Japanese	希望 愛 平和

Figure 5-5 shows the same strings separated into three lists: one for each language. Each list has been sorted independently by applying the sorting rules for a language. The language lists in each script system have been ordered by calling the `LanguageOrder` function.

Figure 5-5 Strings in different languages sorted by language within script

Script	Language	Sorted strings
Roman	English	hope love peace
	German	Frieden Hoffnung Liebe
Japanese	Japanese	希望 愛 平和

Sorting Strings in the Same Language

The Text Utilities provide a number of routines that you can use to compare and sort strings in the same language. Some of these routines perform a comparison that assumes single-byte character codes in the strings; others take into account the sorting rules of the current script system, and still others allow you to explicitly specify the script system resource to use for sorting strings.

Comparing strings can be an extremely intricate operation, because in many languages you may have to account for subtleties such as complex characters, ignorable characters, and exceptional words. Even for a straightforward language such as English, you can’t always determine the sorting order by a simple table lookup or character value comparison.

This section provides an introduction to some of the principles of text comparison and sorting used by the Macintosh script management system. It then describes the routines you can use for different comparison tasks.

Primary and Secondary Sorting Order

Sorting consists of two steps: determining primary sorting order and determining secondary sorting order.

What happens in primary sorting order and secondary sorting order depends on the language of the strings that are being sorted; however, there are two levels of importance in the sorting operation, with some sorting differences subordinate to others. In the primary sorting order for many Roman script system languages, uppercase and lowercase characters are equivalent and diacritical marks are ignored. Thus, after primary sorting, the two strings “The” and “thé” are considered equivalent. In the secondary sorting order, lowercase characters are ranked after uppercase characters and characters with diacritical marks are ranked individually. Thus, after secondary sorting, “The” would sort before “thé”.

You can think of the character ranking that is used to determine sorting order as a two-dimensional table. Each row of the table is a class of all characters such as all *A*’s: uppercase and lowercase, with and without diacritical marks. The characters are ordered within each row to form a secondary sorting order, while the order within each column determines primary sorting order. Table 5-1 shows an example of such a table.

Table 5-1 Excerpt from the Standard Roman script system sorting order

Primary sorting order	Secondary sorting order
A	A À Á Â Ä Å Æ a à á â ä å æ
B	B b
C	C Ç c ç
D	D d
E	E È É Ê Ë e è é ê ë
F	F f

In primary sorting, each of the characters in the first row would be considered equivalent and sorted before characters in the second row. In secondary sorting, the order of the characters in each row would be taken into consideration. Another way of saying this is to say that primary sorting characteristics take precedence over secondary sorting characteristics: if any primary differences are present, all secondary differences are ignored. When the strings being compared are of different lengths, each character in the longer string that does not correspond to a character in the shorter one results in a “comes after” result. This takes precedence over secondary sorting order.

For example, here is a list of strings that have not yet been sorted:

Å ab Ác ac Ab àb Ac

After primary sorting, the list appears as follows

Å ab Ab àb Ác ac Ac

After secondary sorting, the list appears as follows

Å Ab ab àb ac Ác Ac

Expansion and Contraction of Characters

In some languages, a single character may be expanded—that is, sorted as a sequence of characters. First, the sorting routine expands the character, then it performs sorting on the expanded version. Next, the sorting routine recombines the character and then performs secondary sorting. For instance, the *ä* in German may be sorted as if it were the two characters *ae*: *Bäcker* would come after *Bad*, but before *Bahn*.

A sequence of characters may also be contracted—that is, sorted as a single character. For instance, *ch* in Spanish may be sorted as if it were one character that sorts after *c* but before *d*: *coche* comes after *coco* but before *codo*.

Ignorable Characters

Certain characters need to be ignored unless the strings are otherwise equal; that is, these characters have no effect on the primary sorting order, but they do influence the secondary sorting order. In English, hyphens, apostrophes, and spaces are ignorable characters. For instance, the hyphen is ignored in primary sorting order in English: *black-bird* would come after *blackbird*, but before *blackbirds*.

Converting and Stripping Characters

Sometimes you may want to strip out certain characters (notably diacritical marks) or convert the case of characters in a string to produce a different comparison result. For example, you may want to convert all alphabetic characters in two strings into uppercase before comparing the strings, rendering uppercase and lowercase characters equivalent. The Text Utilities provide a number of routines for converting the case of characters and for stripping diacritical marks. These routines are described in the section “Modifying Text,” which begins on page 5-18.

Special Cases for Sorting

Sometimes the sorting order changes drastically for special cases. For instance, when words are understood to be abbreviations, the strings should be sorted as if they were spelled out in full, as in the following examples.

First string	Second string	Explanation
McDonald	Mary	*McDonald* is treated as *MacDonald*
St. James	Smith	*St.* is an abbreviation for *Saint*
Easy Step	Easy St.	*St.* is an abbreviation for *Street*

Cases such as these require a direct dictionary lookup and are not handled automatically by the Macintosh script management system. Note that some abbreviations are context-dependent, such as *St.*, which may denote *Saint* or *Street*, depending on the meaning of the adjacent text.

Variations in Sorting Behavior

Here are some examples of variations in sorting behavior in different writing systems of the world.

- Sorting in Japanese depends upon the subscript. Kana and Romaji sorting are complicated by the presence of both 1- and 2-byte character codes. Moreover, many Katakana symbols have diacritical marks indicating a sound modification. For example, the symbol for *ga i*s formed from the symbol for *ka.* The secondary sorting order for *ga* includes the four combinations of 1-byte or 2-byte *ka* with the 1-byte or 2-byte diacritical mark, plus a 2-byte character that combines the character and diacritical into a single glyph.

  In the Japanese script system, Kanji is currently sorted by character code, which can produce unexpected results. Proper sorting in Kanji is commonly done using one of three methods:
  - First by a character's primary radical, then by the number of remaining strokes.
  - First by the number of total strokes, then by the primary radical.
  - By sound value.
- Sorting in Arabic is quite straightforward except that some characters are ignorable, such as vowels and the extension bar (used to lengthen the cursive connection between characters). Vowels in Arabic are also diacritical marks, overlapping over or under the previous character (the character to the right).
- The Thai script system currently provides for third-level sorting involving character clusters. For more on character clusters, see the chapter "Introduction to Text on the Macintosh" in this book.

Note
If you need to modify a script system's standard string comparison or replace it with your own version, you have to create your own string-manipulation (`'itl2'`) resource by following the guidelines described in the appendix "International Resources" in this book. ◆

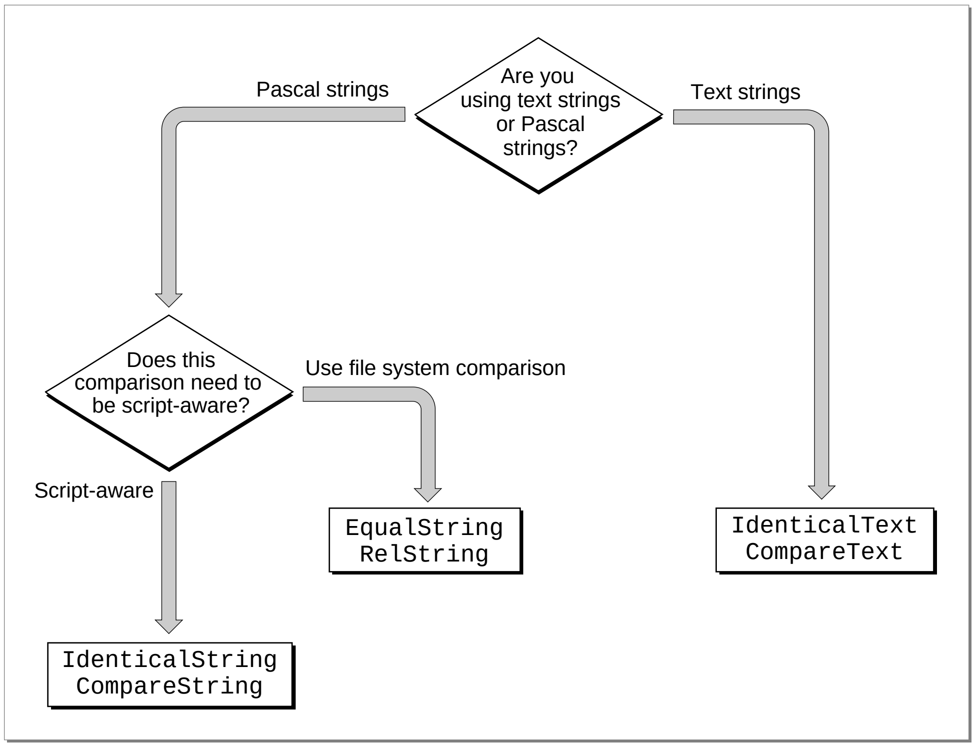
Choosing a Comparison Routine

The Text Utilities include six different routines for comparing one string to another. Three of these routines test two strings for equality and the other three determine the ordering relationship between two strings. You can use these routines with Pascal strings and text strings, and they allow you to work with information from various resources or to ignore script and language information altogether.

Figure 5-6 provides you with convenient guidelines for choosing from among the comparison routines included in the Text Utilities. You first decide whether you are comparing two Pascal strings or two text strings, and then decide whether to unconditionally use the Macintosh file system string comparison rules or to explicitly specify a string-manipulation resource that defines such rules. If you use a routine that requires a parameter for a string-manipulation resource handle, you can specify `NIL` for the value of that parameter to indicate that you want the current script system's string-manipulation resource.

The top routine names in the boxes in Figure 5-6 are used to test two strings for equality, and the bottom routine names in the boxes are used to compare two strings and determine their sorting order. The term script-aware is used in this figure to indicate that you must explicitly specify a string-manipulation resource as a parameter to a routine, rather than the routine automatically using the file system's string-manipulation rules.

Figure 5-6 Choosing a string comparison routine


Are you using text strings or Pascal strings?
Pascal strings
Text strings
Does this comparison need to be script-aware?
Use file system comparison
Script-aware
EqualString
RelString
IdenticalText
CompareText
IdenticalString
CompareString

The Macintosh file system string comparison rules are a subset of the Roman script system comparison rules. These rules are used when the Macintosh file system compares filenames for sorting. Since the *Macintosh character set* only contains characters with codes from $0 to $D8, the file system comparison rules only work correctly for character codes through $D8. You should only use the routines that use these rules when you are trying to emulate the way that the Macintosh file system compares strings.

Table 5-2 describes the sorting behavior implemented by the routines that use the file system comparison rules.

Table 5-2 Sorting features of the Macintosh file system

Reordering of ligatures		**Stripping of diacritical marks**		**Uppercase conversion**	
Ligature	Falls between	Marked character	Stripped to	Lower	Upper
Æ	Å and a	Ä, Å, À, Ã	A	a–z	A–Z*
æ	å and B	Ç	C	à	À
Œ	Ø and o	É	E	ã	Ã
œ	ø and P	Ñ	N	ä	Ä
ß	s and T	Ö, Õ, Ø	O	å	Å
		Ü	U	æ	Æ
		á, à, â, ä, ã, å, ª	a	ç	Ç
		ç	c	é	É
		é, è, ê, ë	e	ñ	Ñ
		í, ì, î, ï	i	ö	Ö
		ñ	n	õ	Õ
		ó, ò, ô, ö, õ, ø, º	o	ø	Ø
		ú, ù, û, ü	u	œ	Œ
		ÿ	y	ü	Ü

*All simple lowercase Roman characters are converted to their uppercase equivalents.

Testing Two Strings for Equality

To test whether two strings are equal, use `EqualString`, `IdenticalString`, or `IdenticalText`. You can use the first two functions with Pascal strings, and the last one with text strings.

The functions that work with Pascal strings—`EqualString` and `IdenticalString`—allow you to specify the kind of information you want to consider in your test. If you want to test two Pascal strings using the Macintosh file system comparison rules, use `EqualString`. If you want to consider the information from a string-manipulation resource, use the `IdenticalString` function. You can explicitly specify the handle of a string-manipulation resource or you can specify `NIL` as the value to indicate that you want the current script's string-manipulation resource used.

To test two text strings for equality, you use the `IdenticalText` function, which makes use of the information in a string-manipulation resource. You can explicitly specify the handle of a string-manipulation resource or you can specify `NIL` as the value to indicate that you want the current script's string-manipulation resource used.

Comparing Two Strings for Ordering

There are also three Text Utilities routines that compare two strings and return a value that indicates whether the first string is less than, equal to, or greater than the second string. Two of these routines take Pascal string parameters and the other takes text string parameters.

To compare one Pascal string to another, you have to choose either the `RelString` function or the `CompareString` function. If you want to compare two Pascal strings using the Macintosh file system comparison rules, use `RelString`. If you want to consider the information from a string-manipulation resource, use the `CompareString` function. You can explicitly specify the handle of a string-manipulation resource or you can specify `NIL` as the value to indicate that you want the current script's string-manipulation resource used.

To compare one text string to another, you use the `CompareText` function, which makes use of the information in a string-manipulation resource. You can explicitly specify the handle of a string-manipulation resource or you can specify `NIL` as the value to indicate that you want the current script's string-manipulation resource used.

Modifying Text

The Text Utilities include a number of routines that you can use to modify the contents of strings. Several of these routines operate on Pascal strings, while others operate on text strings.

Several of the text modification routines also take a script code parameter, which is used to indicate which script system's resources should be used to define the results of various character modifications. Script codes are described in the chapter "Script Manager" in this book.

There are three kinds of text modification routines:

- routines that convert the case of characters and strip diacritical marks from characters in a string
- routines that truncate a string to make it fit into a specified area on the screen
- routines that search for a character pattern in a string and replace it with a different character pattern

Converting Characters and Stripping Marks in Strings

Several Text Utilities routines allow you to convert the case of characters and strip diacritical marks from strings. They can be useful when you want to present strings in a simplified form or to store strings in a form that can increase the efficiency of a comparison.

You can use the `UpperString` procedure to convert any lowercase letters in a Pascal string into their uppercase equivalents; however, this procedure assumes that you are using the Macintosh file system conversion rules and does not use any of the information in the international resources to perform its conversion.

You can use the `UppercaseText` procedure to convert any lowercase letters in a text string into their uppercase equivalents. This procedure takes a script code parameter and uses the case conversion information in the string-manipulation resource for the indicated script system to convert the characters.

The `LowercaseText` procedure converts any uppercase letters in a text string into their lowercase equivalents. This procedure takes a script code parameter and uses the case conversion information in the string-manipulation resource for the indicated script system to convert the characters.

The `StripDiacritics` procedure removes any diacritical marks from a text string. This procedure takes a script code parameter and uses the information in the string-manipulation resource for the indicated script system to determine what character results when a diacritical mark is stripped.

The `UppercaseStripDiacritics` procedure combines the effects of the `UppercaseText` and `StripDiacritics` procedures: it converts any lowercase letters to their uppercase equivalents and strips any diacritical marks from characters in a text string. This procedure also takes a script code parameter, which specifies which script system's resources are used to determine conversion results.

Certain other routines in Macintosh system software convert characters in a text string. The `TransliterateText` function converts characters from one subscript into the closest possible approximation in a different subscript within the same script system. The `IntlTokenize` function converts text into language-independent tokens, for further processing by interpreters or compilers. `TransliterateText` and `IntlTokenize` are documented in the chapter "Script Manager" in this book.

Fitting a String Into a Screen Area

When you want to ensure that a string fits in a certain area of the screen, you can use either the `TruncString` or `TruncText` routine. Each performs the same operation: truncating the string (removing characters from it) so that it fits into a specified pixel width. The `TruncString` function truncates a Pascal string and the `TruncText` function truncates a text string.

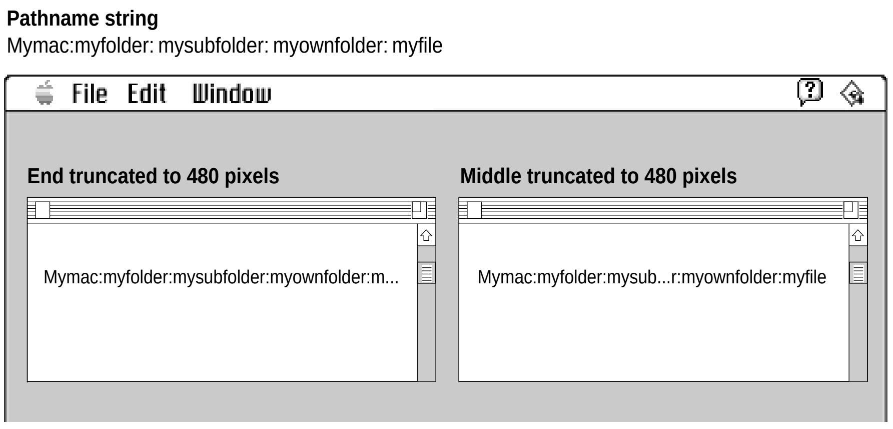
Both of the truncation functions use the current font—the font currently in use in the current graphics port—and its script to determine where the string should be truncated. The font size is used to determine how many characters can completely fit in the number of pixels specified as a parameter to the function.

Both functions also take a parameter that specifies where any needed truncation is to occur. You can specify that characters are to be truncated from the end or from the middle of the string, as MPW does with pathnames, for example.

A truncation indicator is inserted into a string after characters are truncated; in the U.S. Roman script system, the ellipsis (…) is used for this purpose. You should specify the truncation indicator by token, rather than by specific character code, so that the proper indicator is applied to each script system's text. Specify a token from the untoken table of the tokens (`'itl4'`) resource of the script system of the current font. The untoken table is described in the appendix "International Resources" in this book.

Truncating a string in its middle is commonly used on pathnames, where you want the user to see the beginning and end of the full path, but are willing to sacrifice some of the information in the middle, as shown in Figure 5-7.

Figure 5-7 Truncating a pathname in its middle


Pathname string
Mymac:myfolder: mysubfolder: myownfolder: myfile
File
Edit
Window
End truncated to 480 pixels
Mymac:myfolder:mysubfolder:myownfolder:m...
Middle truncated to 480 pixels
Mymac:myfolder:mysub...r:myownfolder:myfile

The code in Listing 5-2 performs the truncation that is illustrated in Figure 5-7. Assuming that each character in the string requires 12 pixels, then 480 pixels will be wide enough to hold 40 characters:

Listing 5-2 Truncating a pathname

```
str := "Mymac:myfolder:mysubfolder:myownfolder:myfile"
ans := TruncString( 480, str, truncEnd );    {480 pixels available}
   {str would be "Mymac:myfolder:mysubfolder:myownfolder:…"}
ans := TruncString( 480, str, truncMiddle );
   {str would now be "Mymac:myfolder:mysub…:myownfolder:myfile"}
```

Since the truncation functions can alter the length and contents of the string that you pass in, it is good practice to make a copy of a string before passing it to one of them.

Replacing a Portion of a String

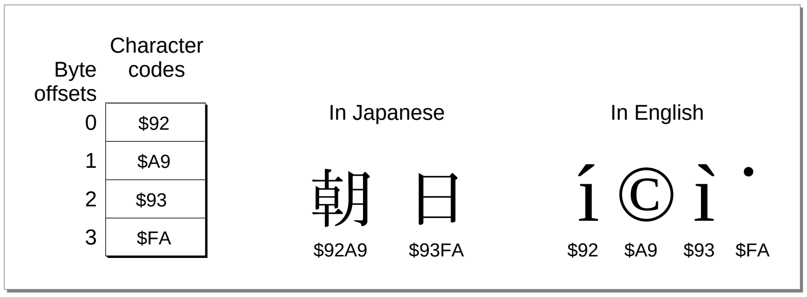
The Text Utilities include two routines for replacing a portion of a string with another string. Each of these routines searches through a string looking for the pattern string. Whenever it finds an occurrence of the pattern string, the routine replaces it with the new string.

The `ReplaceText` function takes information about the current script system into account: it looks through the string character-by-character rather than byte-by-byte. Specifically, this means that `ReplaceText` properly examines strings that contain both 1-byte and 2-byte characters.

The `Munger` function searches for a sequence of bytes and replaces it with another sequence of bytes that you specify. It provides the same capability as `ReplaceText`, but searches for a byte pattern without regard to character length. In a string that contains a mixture of 1-byte and 2-byte characters, `Munger` can, under some conditions, wrongly find a pattern string. This is because the second byte in some 2-byte characters can be wrongly regarded as a 1-byte character.

For example, suppose that you want to search a string for the copyright (“©”) character and replace each occurrence with the string “Registered”. If you use `Munger` to search a string with Japanese characters in it, `Munger` will mistakenly find and replace the byte with value A9, which is really part of a 2-byte character in the Japanese script system. Figure 5-8 shows how the Japanese word for “morning sun” could be incorrectly identified as containing the copyright character.

Figure 5-8 Replacing a portion of a string with 1-byte and 2-byte characters



`Munger` provides a great deal of power, allowing you to perform many interesting substitutions; however, you need to limit your use of `Munger` in applications that are script-aware, or else do your own checking for 2-byte characters.

Listing 5-3 uses the `ReplaceText` and `TruncText` functions. It assumes that you have `Str255` strings containing base text and substitution text and that you want the result to fit in a specified number of pixels.

Listing 5-3 Substituting and truncating text

```
CONST
   maxInt = 32767;
VAR
   baseString: Str255;
   subsString: Str255;
   baseHandle: Handle;
   subsHandle: Handle;
   keyStr: Str15;
   sizeL: LongInt;
   myWidth: Integer;
   length: Integer;
   result: Integer;
   myErr: OSErr;

BEGIN
   baseString:'abcdefghijklmnopqrstuvwxyzabcdefghijklmnopqrstuvwxyz';

   subsString := 'KILROY WAS HERE';    {insert this into baseString}
   keyStr := 'mnop';                   {replace this with subString}
   myWidth := 500;                     {truncate string at this width}
   sizeL := ord(baseString[0]);
   myErr := PtrToHand(@baseString[1], baseHandle, sizeL);
   IF myErr <> noErr
      THEN DoError(myErr);
   sizeL := ord(subsString[0]);
   myErr := PtrToHand(@subsString[1], subsHandle, sizeL);
   IF myErr <> noErr
      THEN DoError(myErr);
   result := ReplaceText(baseHandle, subsHandle, keyStr);
   IF result < 0
      THEN DoError(result);
   sizeL := GetHandleSize(baseHandle);
   IF MemError <> noErr
      THEN DoError(MemError);
   length := sizeL;
   HLock(baseHandle);
   IF MemError <> noErr
      THEN DoError(MemError);          {Memory Manager error}
   result := TruncText(myWidth, baseHandle^, length, TruncEnd);
   IF result < 0
      THEN DoError(result);
   DrawText(baseHandle^, 0, length);
   HUnlock(baseHandle);
   IF MemError <> noErr
      THEN DoError(myErr);             {Memory Manager error}
END;
```

The code in Listing 5-3 first calls the `ReplaceText` function to replace a portion of the base string (the string initialized to contain the alphabet) with another string. Since two of the parameters to `ReplaceText` are string handles, the code first creates handles to the two strings and verifies that no errors occurred. It then calls the `TruncText` function to remove characters from the end of the modified base string so that the string can be displayed, using the text font, size, and style settings in the current graphics port, in an area 500 pixels wide. Once the string is truncated, the code calls the QuickDraw procedure `DrawText` to draw the string in the current graphics port on the screen.

Finding Word, Line, and Script Run Boundaries

This section describes the Text Utilities routines that you can use to determine where the boundaries of the current word in a text sequence are, where to break the line for drawing text, and where the end of the current subscript text run is. These routines are commonly used in word-processing applications.

Finding Word Boundaries

When working with text in your application, you sometimes need to process each word in the text. You can use the `FindWordBreaks` procedure to determine the starting and ending locations in a string of a word. You pass `FindWordBreaks` a string and a starting position, and it searches backward for the start of the word, then searches forward for the end of the word.

This procedure normally uses the string-manipulation (`'itl2'`) resource of the current script system in determining where the word boundaries are. Most string-manipulation resources include a word-selection break table of type `NBreakTable` that specifies what constitutes a word boundary in that script; however, some string-manipulation resources do not include such a table, in which case `FindWordBreaks` uses default definitions of word boundaries. Some script systems provide a separate extension that allows

`FindWordBreaks` to find word breaks in a more sophisticated fashion such as using a dictionary lookup. The format of the word-selection break table is described in the appendix “International Resources” in this book.

This procedure returns the beginning and ending of a word in a string. Theses values are returned in a table of type `OffsetTable`, which contains values that indicate the starting and ending positions in the string of the word. The `OffsetTable` data structure is described in the section “The Offset Table Record” on page 5-44.

You can also use `FindWordBreaks` to break lines of text, although the procedure is more complicated than using `StyledLineBreak`, as described in the next section. For more information, see the discussion of text drawing in the chapter “QuickDraw Text” in this book.

Finding Line Breaks

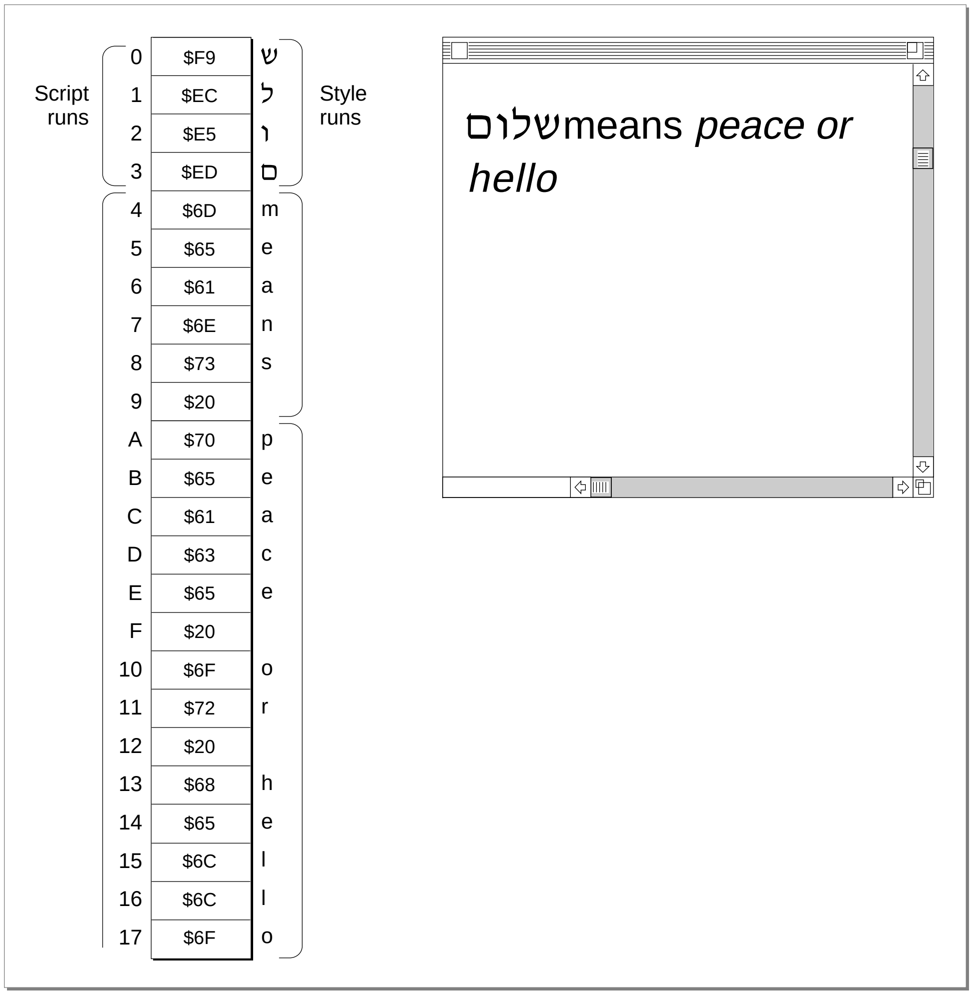
You display text on the Macintosh screen by calling the QuickDraw text routines. These routines handle text in different fonts, styles, and sizes, and even draw text that is displayed in different directions. However, the QuickDraw text display routines do not break lines for you to fit into screen areas of your own designation, which means that you have to display your text line-by-line. (The QuickDraw text routines are described in the chapter “QuickDraw Text” in this book.)

To draw a string line-by-line, you need to use the `StyledLineBreak` function. What you do is start at the first character in your text and use `StyledLineBreak` to search for the first line break, draw that portion of the string, and then start up again with the character that follows the line break. You continue this process until the remaining characters all fit on one line. The size and style of the glyphs are factors in determining how many characters fit onto a line, since they affect the number of pixels required for each glyph on the line. Another factor in breaking lines is that it is desirable to break a line on a word boundary whenever possible.

The `StyledLineBreak` function looks for the next line break in a string. It accommodates different fonts, styles, and glyph sizes, and accounts for complications such as the word boundary rules for the script system of the text. You usually call `StyledLineBreak` to traverse a line in memory order, which is not necessarily the same as display order for mixed-directional text. `StyledLineBreak` finds line breaks on word boundaries whenever possible. `StyledLineBreak` always chooses a line break for the last style run on the line as if all trailing whitespace in that style run would be stripped.

The `StyledLineBreak` function works on one style run at a time. To use `StyledLineBreak`, you must represent the text in your documents in a manner that allows you to quickly iterate through script runs in your text and style runs within each script run. Figure 5-9 shows an example of a line break in a text string with multiscript text runs.

Figure 5-9 Finding line breaks in multiscript text



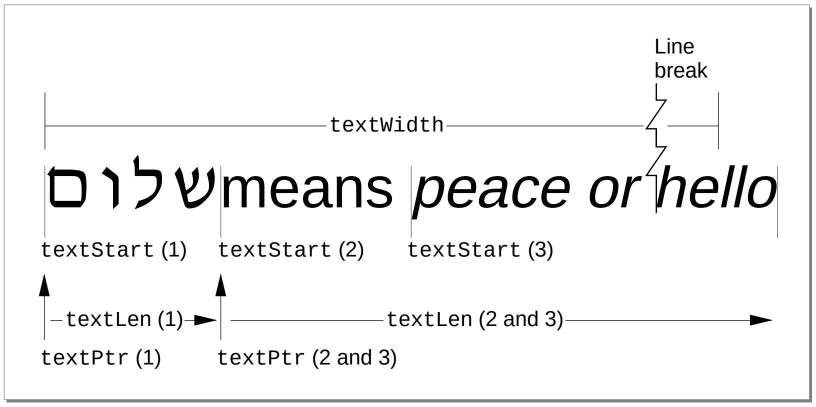
Use the `StyledLineBreak` function when you are displaying text in a screen area to determine the best place to break each displayed line. You can only use this function when you have organized your text in script runs and style runs within each script run. This type of text organization used by most text-processing applications that allow for multiscript text.

What you do is iterate through your text, a script run at a time, using `StyledLineBreak` to check each style run in the script run until the function determines that it has arrived at a line break. As you loop through each style run, before calling `StyledLineBreak`, you must set the text values in the current graphics port that are used by QuickDraw to measure the text. These include the font, font size, and font style of the style run. For details on these parameters, see the chapter "QuickDraw Text" in this book.

Once `StyledLineBreak` has arrived at a line break, you can display the line, advance the pointers into your text, and call the function again to find the next line break. You continue to follow this sequence until you've reached the end of your text. `StyledLineBreak` does not break on a space character, so a sequence of spaces of any length remains with the previous line.

The `StyledLineBreak` function uses a number of parameters; the value of some of these parameters must change for each style run, and the value of others must change for each script run. Figure 5-10 illustrates how the parameters of the `StyledLineBreak` function are used when finding a line break in text that contains a number of script and style runs.

Figure 5-10 Relationships of the parameters of `StyledLineBreak`



The `textPtr` parameter points to the start of the script run, the `textStart` parameter is the location of the start of the style run, and the `textLen` parameter is the number of bytes in the style run. The `textWidth` parameter specifies the number of pixels in the display line. Other parameters are `textEnd`, which specifies the number of bytes in the script run, and `textOffset`, in which the location of the break is returned. Declarations and descriptions of these parameters are found in the section "StyledLineBreak" beginning on page 5-79.

Note that the style runs in `StyledLineBreak` must be traversed in memory order, not in display order. For more information about this, read about the `GetFormatOrder` routine in the chapter "QuickDraw Text" in this book. It is also important to remember that word boundaries can extend across style runs, but cannot extend across script runs.

The `StyledLineBreak` function looks for a line break on a word boundary. The only time it cannot find such a break is when a word spans across an entire line. If such a word starts past the beginning of the line, `StyledLineBreak` determines that a break should occur before the start of the word; otherwise, it breaks the line in the middle of the word, at a character boundary instead of at a word boundary. `StyledLineBreak` uses the value of the `textOffset` parameter to differentiate between these two cases. The `textOffset` parameter must be nonzero for the first call on a line and zero for each subsequent call to the function on the line.

No matter which case occurs, `StyledLineBreak` returns a code that specifies whether or not it found a break and what kind of break (word or character boundary) it is. This value is one of the constants defines as the `StyledLineBreakCode` type:

`StyledLineBreak` constant	**Value**	**Meaning**
`BreakOverflow`	2	No break is necessary because the current style run fits on the line (within the width)
`BreakChar`	1	Line breaks on character boundary
`BreakWord`	0	Line breaks on word boundary

`StyledLineBreak` automatically decrements the `textWidth` variable by the width of the style run for use on the next call. You need to set the value of `textWidth` before calling it to process a line. Listing 5-4 shows a basic loop structure that you can use to call `StyledLineBreak` in your application.

Listing 5-4 Using the `StyledLineBreak` function

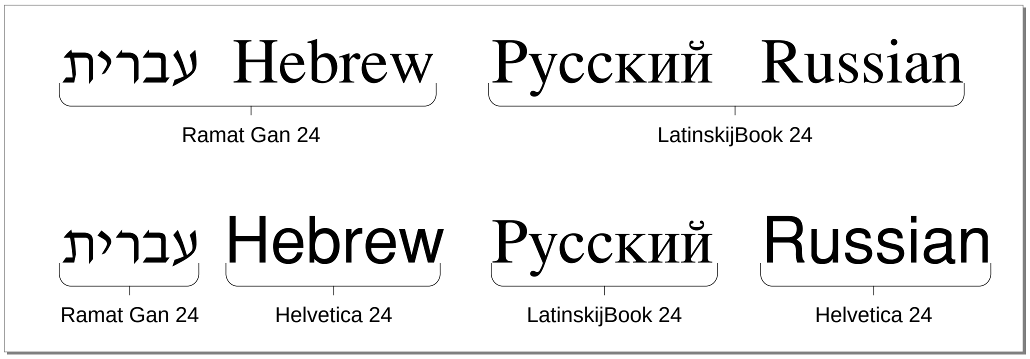
```
REPEAT                            {repeat for each line}
   textOffset := 1
   textWidth := number of display pixels available for line
   done := FALSE;
   WHILE not done DO
      BEGIN                       {for each script run}
      textPtr := the address of the first byte of the script run
      textLen := the number of bytes in the script run
      WHILE not done DO
         BEGIN                    {for each style run}
         textStart := byte offset within script run of the start
                     of the style run
         textEnd := byte offset within script run of the end
                     of the style run
         {Set up the QuickDraw font parameters for style run}
         ...
         ans := StyledLineBreak(textPtr, textLen, textStart,
                        textEnd, flags, textWidth, textOffset);
         if ans <> smBreakOverflow
            THEN done := TRUE;
            ELSE textOffset := 0;{always 0 after first call}
         END;
      END;
      {Display the text that starts at textPtr & continues }
      { for textOffset bytes}
      ...
UNTIL                            {until no more text to process}
```

Finding Subscripts Within a Script Run

Some script systems include subscripts, which are character sets that are subsidiary to the main character set. One useful subscript is the set of all character codes that have the same meaning in Roman as they do in a non-Roman script. For other scripts such as Japanese, there are additional useful subscripts. For example, a Japanese script system might include some Hiragana characters that are useful for input methods.

When you are displaying or working with a string that contains subscript characters, it is often convenient to identify the subscript text runs so that you can treat those characters differently. You might, for instance, want to display the Roman subscript text in a different font, or apply different rules to it when searching for word boundaries. In Figure 5-11, the English words "Hebrew" and "Russian" are initially drawn in native language fonts from their script systems. Each of these words is then extracted and redrawn using a font from the Roman script system.

Figure 5-11 Extracting blocks of Roman text



The `FindScriptRun` function is used to identify blocks of subscript text in a string. `FindScriptRun` searches a string for such a block, and sets a `VAR` parameter to the length in bytes of the subscript run that begins with the first character in the string.

`FindScriptRun` also returns a script-run status record, which specifies the script code and subscript information for the block of text. The fields of the script-run status record are described in the section “FindScriptRun,” beginning on page 5-81.

Working With Date and Time Strings

Applications that address international audiences must work with how the numeric-format (`'itl0'`) resource and long-date-format (`'itl1'`) resource handle the differences in date and time formats used in different countries and regions of the world.

The numeric-format resource contains general conventions for formatting numeric strings. It provides several different definitions, including separators for decimals, thousands, and lists; currency information; time values; and short date formats. Some of the variations in date and time formats are shown in Table 5-3.

Table 5-3 Variations in time and short date formats

Morning	Afternoon	Short date	System software
1:02 AM	1:02 PM	2/1/90	United States
1:02	13:02	02/01/90	Canadian French
1:02 AM	1:02 PM	90.01.02	Chinese
1:02	13:02	02-01-1990	Dutch
1:02 Uhr	13:02 Uhr	2.1.1990	German
1:02	13:02	2-01-1990	Italian
01.02	13.02	90-02-01	Swedish

For time and date values, the numeric-format resource includes values that specify this information:

- the order of the month, day, and year values in short date formats
- which separator to use in the short date format (for example, : or / or -)
- the trailing string to display for morning (for example, *A.M.*)
- the trailing string to display for evening (for example, *P.M.*)
- up to 4 trailing bytes to display for 24-hour times before noon, and another 4 bytes to display for 24-hour times at noon and after. For example, the German string *Uhr* is used for both purposes.
- whether or not to indicate leading zeros in each of the time elements (hours, minutes, and seconds)

The long-date-format resource includes conventions for long date formats, abbreviated date formats, and the regional version of the script the resource is associated with. Some of the variations in long and abbreviated date formats are shown in Table 5-4.

Table 5-4 Variations in long and abbreviated date formats

Long date	Abbreviated date	System software
Tuesday, January 2, 1990	Tue, Jan 2, 1990	United States
Tuesday, 2 January 1990	Tue, 2 Jan 1990	Australian
Mardi 02 janvier 1990	Mard 02 janv 1990	Canadian French
tirsdag 2. januar 1990	tir 2. jan 1990	Danish
Mardi 2 Janvier 1990	Mar 2 Jan 1990	French

The long-date-format resource includes values that specify this information:

- the names of the days
- the names of the months
- which punctuation to use for abbreviated day names and month names

You can optionally add an extension to a long-date-format resource that adds a number of other specification capabilities, including the following:

- a calendar code for the specification of calendars other than the standard Gregorian calendar, such as the Arabic calendar
- a list of extra day names for calendars with more than seven days
- a list of extra month names for calendars with more than twelve months
- a list of abbreviated day names
- a list of abbreviated month names
- a list of additional date separators

Many of the Apple-supplied long-date-format resources already include such extensions.

The Text Utilities routines that work with dates and times use the information in the long-date-format and numeric-format resources to create different string representations of date and time values. The Macintosh Operating System provides routines that return the current date and time to you in numeric format; you can then use the Text Utilities routines to convert those values into strings that can be presented in different international formats.

The Text Utilities also include routines that can parse date and time strings as entered by users and fill in the fields of a record with the components of the date and time, including the month, day, year, hours, minutes, and seconds.

For more details on the numeric-format (`'itl0'`) and long-date-format (`'itl1'`) resources, see the appendix “International Resources” in this book. For information on obtaining the current date and time values from the Macintosh Operating System, see *Inside Macintosh: Operating System Utilities*.

Converting Formatted Date and Time Strings Into Internal Numeric Representations

When your application works with date and time values, it must convert string versions of dates and times into internal numeric representations that it can manipulate. You might, for example, need to convert a date typed by the user into a numeric representation so that you can compute another date some number of days ahead. You can then format the new value for display as a formatted date string.

The Text Utilities contains two routines that you can use to parse formatted date and time values from input strings and create an internal numeric representation of the date and time. The `StringToDate` function parses an input string for a date, and the `StringToTime` function parses an input string (possibly the same input string) for time information.

Both of these functions pass a date cache record as one of the parameters. A date cache record is a data structure of type `DateCacheRec` that you must declare in your application. Because you must pass this record as a parameter, you must initialize it by calling the `InitDateCache` function before calling `StringToDate` or `StringToTime`. You need to call `InitDateCache` only once—typically in your main program initialization code. For more information about the date cache record and the `InitDateCache` function, see the section "InitDateCache" on page 5-83.

Both the `StringToDate` and the `StringToTime` functions fill in fields in a long date-time record, which is defined by a `LongDateRec` data structure. This data type is described in the book *Inside Macintosh: Operating System Utilities*.

You usually use `StringToDate` and `StringToTime` sequentially to parse the date and time values from an input string and fill in these fields. Listing 5-5 shows how to first call `StringToDate` to parse the date, then offset the starting address of the string, and finally, call `StringToTime` to parse the time.

Listing 5-5 Using `StringToDate` and `StringToTime`

```
str := "March 27, 1992 08:14 p.m.";

strPtr := ptr(ord(@str) + 1); {Pointer to 1st char of str}
strLen := length(str);
status := StringToDate(strPtr, strLen, myDateCache,
                                       numBytes, lDateRec);
strPtr := ptr(ord(@str)+numBytes+1);
strLen := strLen - numBytes;
status := StringToTime(strPtr, strLen, myDateCache,
                                       numBytes, lDateRec);
```

`StringToDate` parses the text string until it has finished finding all date information or until it has examined the number of bytes specified by `textLen`. It returns a status value that indicates the confidence level for the success of the conversion. `StringToDate` recognizes date strings in many formats, including “September 1, 1987,” “1 Sept 1987,” “1/9/1987,” and “1 1987 sEpT.”

Note that `StringToDate` fills in only the year, month, day, and day of the week; `StringToTime` fills in the hour, minute, and second. You can use these two routines sequentially to fill in all of the values in a `LongDateRec` record.

`StringToDate` assigns to its `lengthUsed` parameter the number of bytes that it uses to parse the date; use this value to compute the starting location of the text that you can pass to `StringToTime`.

`StringToDate` interprets the date and `StringToTime` interprets the time based on values that are defined in the long-date-format (`'itl1'`) resource. These values, which include the tokens used for separators and the month and day names, are described in the appendix “International Resources” in this book.

`StringToDate` uses the `IntlTokenize` function, as described in the chapter “Script Manager” in this book, to separate the components of the strings. It assumes that the names of the months and days, as specified in the international long-date-format resource, are single alphanumeric tokens.

When one of the date components is missing, such as the year, the current date value is used as a default. If the value of the input year is less than 100, `StringToDate` determines the year as follows.

1. If (current year) MOD 100 is greater than or equal to 90 and the input year is less than or equal to 10, the input year is assumed to be in the next century.
2. If (current year) MOD 100 is less than or equal to 10 and the input year is greater than or equal to 90, the input year is assumed to be in the previous century.
3. Otherwise, the input year is assumed to be in the current century.

If the value of the input year is between 100 and 1000, then 1000 is added to it. Thus the dates 1/9/87, 1/9/987, and 1/9/1987 are equivalent.

Both `StringToDate` and `StringToTime` return a value of type `StringToDateStatus`, which is a set of bit values that indicate confidence levels, with higher numbers indicating low confidence in how closely the input string matched what the routine expected. Each `StringToDateStatus` value can contain a number of the possible bit values that have been OR’ed together. For example, specifying a date with nonstandard separators may work, but it returns a message indicating that the separator was not standard.

The possible values of this type are described in Table 5-5.

Table 5-5 `StringToDateStatus` values and their meanings

`StringToDateStatus` value	Result of the conversion
`fatalDateTime`	A fatal error occurred during the parse.
`tokenErr`	The token processing software could not find a token.
`cantReadUtilities`	The resources needed to parse the date or time value could not be read.
`dateTimeNotFound`	A valid date or time value could not be found in the string.
`dateTimeInvalid`	The start of a valid date or time value was found, but a valid date or time value could not be parsed from the string.
`longDateFound`	A valid long date was found. This bit is not set when a short date or time was found.
`leftOverChars`	A valid date or time value was found, and there were characters remaining in the input string.
`sepNotIntlSep`	A valid date or time value was found; however, one or more of the separator characters in the string was not an expected separator character for the script system in use.
`fieldOrderNotIntl`	A valid date or time value was found; however, the order of the fields did not match the expected order for the script system in use.
`extraneousStrings`	A valid date or time value was found; however, one or more unparsable strings was encountered and skipped while parsing the string.
`tooManySeps`	A valid date or time value was found; however, one or more extra separator characters was encountered and skipped while parsing the string.
`sepNotConsistent`	A valid date or time value was found; however, the separator characters did not consistently match the expected separators for the script system in use.

For example, if `StringToDate` and `StringToTime` successfully parse date and time values from the input string and more characters remain in the string, then the function result will be the constant `leftOverChars`. If `StringToDate` discovers two separators in sequence, the parse will be successful and the return value will be the constant `tooManySeps`. If `StringToDate` finds a perfectly valid short date, it returns the value `noErr`; if `StringToDate` finds a perfectly valid long date, it returns the value `longDateFound`.

Date and Time Value Representations

The Macintosh Operating System provides several different representations of date and time values. One representation is the standard date-time value that is returned by the Macintosh Operating system routine `GetDateTime`. This is a 32-bit integer that represents the number of seconds between midnight, January 1, 1904 and the current time. Another is the date-time record, which includes integer fields for each date and time component value.

The Macintosh Operating System also provides two data types that allow for longer spans of time than do the standard date-time value and date-time record: the long date-time value and the long-date record. The long date-time value, of data type `LongDateTime`, is a 64-bit, signed representation of the number of seconds since Jan. 1, 1904, which allows for coverage of a much longer span of time (approximately 30,000 years) than does the standard date-time representation. The long date-time record, of data type `LongDateRec`, is similar to the date-time record, except that it adds several additional fields, including integer values for the era, the day of the year, and the week of the year.

The Macintosh Operating System provides four routines for converting among the different date and time data types:

- `DateToSeconds`, which converts a date-time record into a standard date-time value
- `SecondsToDate`, which converts a standard date-time value into a date-time record
- `LongDateToSeconds`, which converts a long-date record into a long date-time value
- `LongSecondsToDate`, which converts a long date-time value into a long-date record

The standard date-time value, the long date-time value, and each of the data structures and routines mentioned in this section are described in the book *Inside Macintosh: Operating System Utilities*.

Converting Standard Date and Time Values Into Strings

When you want to present a date or time value as a string, you need to convert from one of the numeric date-time representations into a formatted string. The Text Utilities include the `DateString` and `TimeString` procedures for converting standard date-time values into formatted strings, and the `LongDateString` and `LongTimeString` procedures for converting long date-time values into formatted strings. Each of these routines uses information from a long-date-format or numeric-format resource that you specify as a parameter.

When you use the `DateString` and `LongDateString` procedures, you can request an output format for the resulting date string. The output format can be one of the three values of the `DateForm` enumerated data type:

```
DateForm = (shortDate,longDate,abbrevDate);
```

Here are examples of the date strings that these specifications produce.

Value	Date string produced
`shortDate`	1/31/92
`abbrevDate`	Fri, Jan 31, 1992
`longDate`	Friday, January 31, 1992

When you request a long or abbreviated date format, the formatting information in a long-date-format resource is used. For short date formats, the information is found in a numeric-format resource. The `DateString` and `LongDateString` procedures use the long-date-format or numeric-format resource that you specify. If you request a long or abbreviated date format, you must include the handle to a long-date-format resource, and if you request a short date format, you must include the handle to a numeric-format resource. If you specify `NIL` for the value of the resource handle parameter, both routines uses the appropriate resource from the current script.

When you use the `TimeString` and `LongTimeString` procedures to produce a formatted time string, you can request an output format for the resulting string. You specify whether or not you want the time string to include the seconds by passing a Boolean parameter to these procedures.

Value	Time string produced
`FALSE`	03:24 P.M.
`TRUE`	03:24:17 P.M.

The `TimeString` and `LongTimeString` procedures use the time formatting information in the numeric-format resource that you specify. This information defines which separator to use between the elements of the time string, which suffix strings to use, and whether or not to add leading zeros in each of the time elements. If you specify `NIL` in place of a resource handle, these procedures use the numeric-format resource from the current script.

Working With Numeric Strings

When you present numbers to the user, or when the user enters input numbers for your application to use, you need to convert between the internal numeric representation of the number and the output (or input) format of the number. The Text Utilities provide several routines for performing these conversions. Some of these routines take into account the many variations in numeric string formats (output formats) of numbers in different regions of the world.

If you are converting integer values into numeric strings or numeric strings into integer values, and you don't need to take international number formats into account, you can use the two basic number conversion routines: `NumToString`, which converts an integer into a string, and `StringToNum`, which converts a string into an integer. These routines are described in the section "Converting Between Integers and Numeric Strings," which begins on page 5-38.

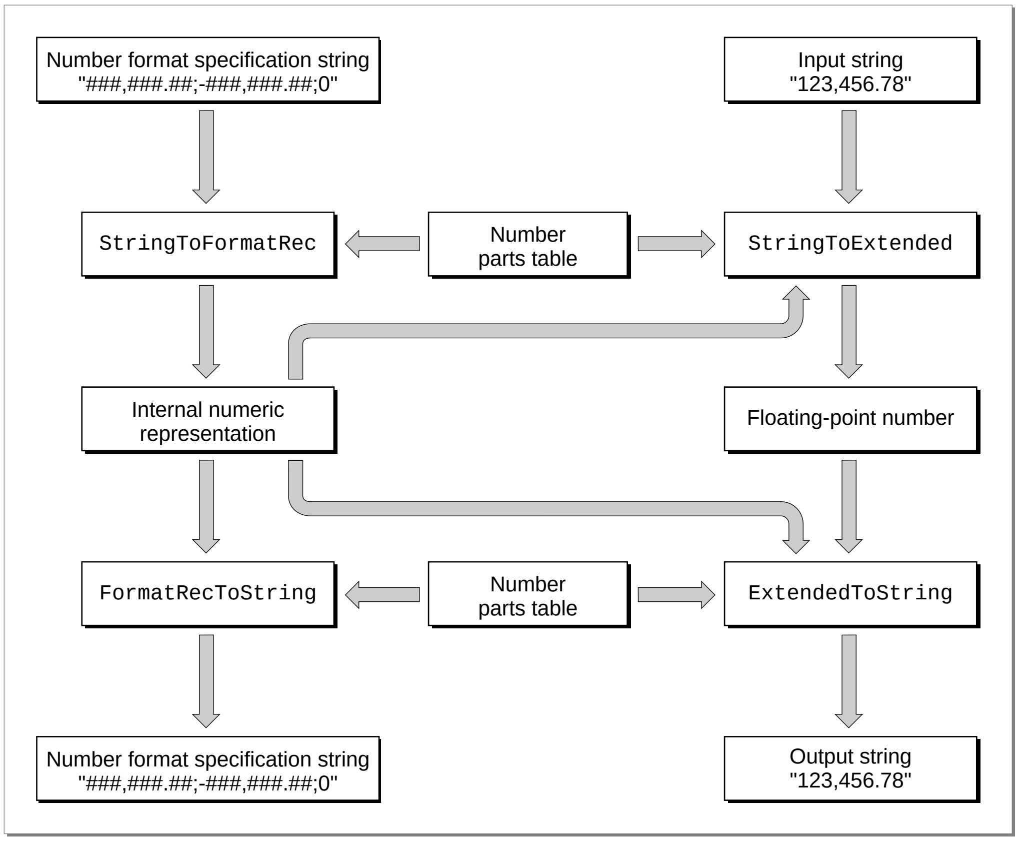
If you are working with floating-point numbers, or if you want to accommodate the possible differences in output formats for numbers in different countries and regions of the world, you need to work with number format specification strings. These are strings that specify the input and output formats for numbers and allow for a tremendous amount of flexibility in displaying numbers.

To use number format specification strings and convert numbers, you need to follow these steps:

1. You first define the format of numbers with a number format specification string. An example of such a string is `###,###.##;-###,###.##;0`. This string specifies three number formats: for positive number values, for negative number values, and for zero values. The section "Using Number Format Specification Strings," which begins on page 5-39, describes these definitions in detail.
2. You must also define the syntactic components of numeric string formats using a number parts table. This table is part of the tokens (`'itl4'`) resource for each script system. It includes definitions of which characters are used in numeric strings for displaying the percent sign, the minus sign, the decimal separator, the less than or equal sign, and other symbols. The number parts table is described with the tokens resource in the appendix "International Resources" in this book.
3. You then use Text Utilities routines to convert the number format specification string into an internal representation that is stored in a `NumFormatStringRec` record. This is a private data type that is used by the number conversion routines. You convert a number format specification string into a `NumFormatStringRec` record with the `StringToFormatRec` function, and you perform the opposite conversion with the `FormatRecToString` function. Both of these functions are described in the section "Converting Number Format Specification Strings Into Internal Numeric Representations," which begins on page 5-43.
4. Once you have a `NumFormatStringRec` record that defines the format of numbers for a certain country or region, you can convert floating-point numbers into numeric strings and numeric strings into floating-point numbers. The `StringToExtended` and `ExtendedToString` functions perform these conversions; these are described in the section "Using Number Format Specification Strings," which begins on page 5-39.

To accommodate all of the possibilities for the different number formats used in different countries and regions, you need to work with numeric strings, number parts tables, number format specification strings, and floating-point numbers. The Text Utilities include the routines shown in Figure 5-12 to make it possible for your application to accept and display numeric strings in many different formats. You can accept an input string in one format and create an output numeric string that is appropriate for an entirely different area of the world. Figure 5-12 summarizes the relationships among the different data and routines used for these conversions.

Figure 5-12 Using the number formatting routines



The number format specification string in the upper left box in Figure 5-12 defines how input and output numeric strings are formatted; in this case, they are formatted in the style most commonly used in the United States, with a comma as the thousand separator. The `StringToFormatRec` format takes the number format specification string as input, along with a number parts table, and creates an internal representation, which is stored in a record of data type `NumFormatStringRec`.

If you later want to create a number format specification string from the internal representation, you can call the `FormatRecToString` function. This function takes a record of type `NumFormatStringRec` and a parts table, and creates a string that you can display or edit.

Once you have an internal representation of your formatting specification, you can use it for converting between strings and floating-point numbers. The `StringToExtended` function takes an input string, a `NumFormatStringRec`, and a number parts table, and creates a floating-point number. The `ExtendedToString` function takes a floating-point number, a `NumFormatStringRec`, and a number parts table, and creates a string.

Each of the four functions shown in Figure 5-12 returns a result of type `FormatStatus`, which is an integer value. The low byte of the result is of type `FormatResultType`, the values of which are summarized in Table 5-6.

Table 5-6 `FormatResultType` values for numeric conversion functions

`FormatStatus` value	Result of the conversion
`fFormatOK`	The format of the input value is appropriate and the conversion was successful.
`fBestGuess`	The format of the input value is questionable; the result of the conversion may or may not be correct.
`fOutOfSynch`	The format of the input number string did not match the format expected in the number format specification string.
`fSpuriousChars`	There are extra characters in the input string.
`fMissingDelimiter`	A delimiter is missing in the input string.
`fExtraDecimal`	An extra decimal was found in the input number string.
`fMissingLiteral`	The close of a literal is missing in the input number string.
`fExtraExp`	There is an extra exponent in the input number string.
`fFormatOverflow`	The number in the input string exceeded the magnitude allowed for in the number format specification.
`fFormStrIsNAN`	The format specification string is not valid.
`fBadPartsTable`	The parts table is not valid.
`fExtraPercent`	There is an extra percentage symbol in the input number string.
`fExtraSeparator`	There was an extra separator in the input number string.
`fEmptyFormatString`	The format specification string was empty.

Converting Between Integers and Numeric Strings

The simplest number conversion tasks for your application involve integer values and do not take international output format differences into account. Text Utilities provides one routine to convert an integer value into a numeric string and another to convert a numeric string into an integer value.

The `NumToString` procedure converts a long integer value into a string representation of it as a base-10 value. The `StringToNum` procedure performs the opposite operation, converting a string representation of a number into a long integer value. For example, Listing 5-6 converts a number into a string and then back again.

Listing 5-6 Converting a long integer into a numeric string

```
VAR
   str: Str255;
   i,j: LongInt;
BEGIN
   i := 4329;
   NumToString(i, str);          {str is now “4329”}
   StringToNum(str, j);          {j is now 4329 }
END;
```

Using Number Format Specification Strings

When you want to work with floating-point values and numeric strings, you need to take into account the different formats that are used for displaying numbers in different countries and regions of the world. Table 5-7 shows some of the numeric string formats that are used in different versions of system software.

Table 5-7 Numeric string formats

Numeric string	System software
1,234.56	All versions
1 234,56	French and others
1.234,56	Danish and others
1 234.56	Greek
1.234 56	Russian
1’234.56	Swiss French, Swiss German

You use number format specification strings to define the appearance of numeric strings in your application. Each number format specification string contains up to three parts: the positive number format, the negative number format, and the zero number format. Each format is applied to a numeric value of the corresponding type: when a positive value is formatted, the positive format is used, when a negative value is formatted, the negative format is used, and when a zero value is formatted, the zero format is used. When a number format specification string contains only one part, that part is used for all values. When a number format specification string contains two parts, the first part is used for positive and zero values, and the second part is used for negative values.

Table 5-8 shows several different number format specification strings, and the output numeric string that is produced by applying each format to a numeric value.

Table 5-8 Examples of number format specification strings

Number format specification string	Numeric value	Output format
`###,###.##;-###,###.##;0`	123456.78	`123,456.78`
`###,###.0##,###`	1234	`1,234.0`
`###,###.0##,###`	3.141592	`3.141,592`
`###;(000);^^^`	–1	`(001)`
`###.###`	1.234999	`1.235`
`###'CR';###'DB';'‘zero’'`	1	`1CR`
`###'CR';###'DB';'‘zero’'`	0	`‘zero’`
`##%`	0.1	`10%`

The three portions of a number format specification string (positive, negative, and zero formats) are separated by semicolons. If you do not specify a format for negative values, negative numbers are formatted using the positive format and a minus sign is inserted at the front of the output string. If you do not specify a format for zero values, they are presented as a single ‘0’ digit.

These number format specification strings contain different elements:

- number parts separators for specifying the decimal separator and the thousand separator
- literals that you want included in the output formats
- digit placeholders
- quoting mechanisms for handling literals correctly
- symbol and sign characters

Number parts separators come in two types: the decimal separator and the thousand separator. In the U.S. localized version of the Roman script system, the decimal separator is the ‘.’ character and the thousand separator is the ‘,’ character. Some script systems use other characters for these separators. The number conversion routines each take a number parts table parameter that includes definitions of the separator characters.

Literals in your format strings can add annotation to your numbers. Literals can be strings or brackets, braces, and parentheses, and must be enclosed in quotation marks. Table 5-9 shows some examples of using literals in number format specification strings.

Table 5-9 Literals in number format strings

Number format specification string	Numeric value	Output format
`###'CR';###'DB';\'“zero\'”`	`1`	`1CR`
`[###' Million '###' Thousand '###]`	`300`	`[300]`
`[###' Million '###' Thousand '###]`	`3210432`	`[3 Million 210 Thousand 432]`

Digit placeholders that you want displayed in your numeric strings must be indicated by digit symbols in your number format specification strings. There are three possible digit symbols: zero digits (0), skipping digits (#), and padding digits (^). The format string in line 4 of Table 5-8 contains examples of each. The actual characters used for denoting each of these are defined in the tokens (`'itl4'`) resource number parts table.

- Zero digits add leading zeros wherever an input digit is not present. For example, –1 in line 4 of Table 5-8 produces (001) because the negative number format is specified as “(000)”, meaning that the output is enclosed in parentheses and leading zeros are added to produce three digits.
- Skipping digits only produce output characters when an input digit is present. For example, if the positive number format is “`###`” and the input string is “1”, then the output format is “1” (not “ 1” as you might expect. Each skipping digit in the number format specification string is replaced by a digit character if one is present, and is not replaced by anything (is skipped) if a digit character is not present.
- Padding digits are like zero digits except that a padding character such as a nonbreaking space is used instead of leading zeros to “pad” the output string. You can use padding digits to align numbers in a column. The number conversion routines each take a number parts table parameter that includes definitions of padding characters.

You must specify the maximum number of digits allowed in your formats, as the number formatting routines do not allow extension beyond them. If the input string contains too many digits, an error (`formatOverflow`) will be generated. If the input string contains too many decimal places, the decimal portion is automatically rounded. For example, given the format of `###.###`, a value of 1234.56789 results in an error condition, and a value of 1.234999 results in the rounded-off 1.235 value.

The number formatting routines always fill in integer digits from the right and decimal places from the left. This can produce the results shown in Table 5-10, which includes a literal in the middle of the format strings to demonstrate this behavior.

Table 5-10 Filling digits in

Number format specification string	Numeric value	Output format
`###'my'###`	`1`	`1`
`###'my'###`	`123`	`123`
`###'my'###`	`1234`	`1my234`
`0.###'my'###`	`0.1`	`0.1`
`0.###'my'###`	`0.123`	`1.123`
`0.###'my'###`	`0.1234`	`0.123my4`

Quoting mechanisms allow you to enclose most literals in single quotation marks in your number format specification strings. If you need to include single quotation marks as literals in your output formats, you can precede them with the escape character (\). Table 5-11 shows several examples of using quoting mechanisms.

Table 5-11 Quoting mechanisms in number format strings

Number format specification string	Numeric value	Output format
`###'CR';###'DB';'‘zero’'`	`1`	`1CR`
`###'CR';###'DB';'‘zero’'`	`-1`	`1DB`
`###'CR';###'DB';'‘zero’'`	`0`	`‘zero’`

Symbol and sign characters in your number format specification strings allow you to display the percent sign, exponents, and numbers’ signs. The actual glyphs displayed for these symbols depend on how they are defined in the number parts table of a tokens resource. The symbols that you can use and the characters used for them in the U.S. Roman script system are shown in Table 5-12.

Table 5-12 Symbols in number format strings

Symbol	U.S. Roman	Number format string	Example
Plus sign	+	`+###`	+13
Minus sign	–	`-###`	–243
Percent sign	%	`##%`	14%
EPlus	E+	`##.####E+0`	1.2344E+3
EMinus	E–	`#.#E-#`	1.2E–3

For more information about these symbols and the tokens defined for them, see the section on number parts tables in the appendix "International Resources" in this book.

Converting Number Format Specification Strings Into Internal Numeric Representations

To use a number format specification string in your application, you must first convert the specification string into an internal numeric representation that is independent of country, language, and other cultural considerations. This allows you to map the number into different output formats. This internal representation is sometimes called a canonical number format. The internal representation of format strings is stored in a `NumFormatStringRec` record.

You can use the `StringToFormatRec` function to convert a number format specification string into a `NumFormatStringRec` record. To perform this conversion, you must also specify a number parts table from a numeric-format resource. The number parts table defines which characters are used for certain purposes (such as separating parts of a number) in the format specification string.

You can use the `FormatRecToString` function to convert a `NumFormatStringRec` record back into a number format specification string, in which the three parts (positive, negative, and zero) are separated by semicolons. This function also uses a number parts table to define the components of numbers; by using a different table than was used in the call to `StringToFormatRec`, you can produce a number format specification string that specifies how numbers are formatted for a different region of the world. You use `FormatRecToString` when you want to display the number format specification string to a user for perusal or modification.

Converting Between Floating-Point Numbers and Numeric Strings

Once you have a `NumFormatStringRec` record that defines the format of numbers for a certain region of the world, you can convert between floating-point numbers and numeric strings.

You can use the `StringToExtended` function to convert a numeric string into an 80-bit floating-point value. `StringToExtended` uses a `NumFormatStringRec` record and a number parts table to examine and convert the numeric string into a floating-point value.

The `ExtendedToString` function performs the opposite conversion: it uses a `NumFormatStringRec` record and a number parts table to convert an 80-bit floating-point value into a numeric string that is formatted for output.

Text Utilities Reference

This section describes the data structures and routines that comprise the Text Utilities. The "Data Structures" section provides a description of the data structures that are used with certain of the Text Utilities routines. The "Routines" section describes the routines you can use in your applications to work with strings.

Data Structures

This section describes the data structures that are used with the Text Utilities routines. Each is used with one or more of the Text Utilities routines to pass information into or to receive information back from the routine.

The Offset Table Record

The `FindWordBreaks` procedure uses the offset table, which is defined by the `OffsetTable` data type. You pass a record of this type by `VAR` to `FindWordBreaks`, and it fills in the fields to specify the location of the next word in the input string. The `FindWordBreaks` procedure is described in the section "FindWordBreaks," which begins on page 5-77.

```
OffsetTable = ARRAY [0..2] of OffPair;

OffPair =
RECORD
   offFirst: Integer;   {offset of first word boundary}
   offSecond: Integer;  {offset of second word boundary}
END;
```

Field descriptions

`offFirst`	The offset in bytes from the beginning of the string to the first character of the word.
`offSecond`	The offset in bytes from the beginning of the string to the last character of the word.

Although the offset table contains three `OffPair` records, the `FindWordBreaks` procedure fills in only the first of these records with the offset values for the word that it finds. The other two entries are for use by the `HiliteText` procedure, which is described in the chapter "QuickDraw Text" in this book.

The Date Cache Record

The `StringToDate` and `StringToTime` functions use the date cache, defined by the `DateCacheRecord` data type, as an area to store date conversion data that is used by the date conversion routines. This record must be initialized by a call to the `InitDateCache` function, which is described in the section "InitDateCache" beginning on page 5-83. The data in this record is private—you should not attempt to access it.

```
DateCachePtr = ^DateCacheRecord;

DateCacheRecord =
PACKED RECORD
      hidden: ARRAY [0..255] OF INTEGER;{only for temporary use}
END;
```

Field descriptions

`hidden`	The storage used for converting dates and times.

The Number Format Specification Record

Four of the numeric string functions use the number formatting specification, defined by the `NumFormatStringRec` data type: `StringToFormatRec`, `FormatRecToString`, `StringToExtended`, and `ExtendedToString`. The number format specification record contains the data that represents the internal number formatting specification information. This data is stored in a private format.

```
NumFormatStringRec =
PACKED RECORD
   fLength: Byte;
   fVersion: Byte;
   hidden: ARRAY [0..253] OF INTEGER;{only for temporary use}
END;
```

Field descriptions

`fLength`	The number of bytes (plus 1) in the hidden data actually used for this number formatting specification.
`fVersion`	The version number of the number formatting specification.
`hidden`	The data that comprises the number formatting specification.

The Triple Integer Array

The `FormatRecToString` function uses the triple-integer array, defined by the `TripleInt` data type, to return the starting position and length in a string of three different portions of a formatted numeric string: the positive value string, the negative value string, and the zero value string. Each element of the triple integer array is an `FVector` record.

```
TripleInt =
ARRAY[0..2] OF FVector;    {indexed by fPositive..fZero}

FVector =
RECORD
   start: Integer;
   length: Integer;
END;
```

Field descriptions

`start`	The starting byte position in the string of the specification information.
`length`	The number of bytes used in the string for the specification information.

Each of the three `FVector` entries in the triple integer array is accessed by one of the values of the `FormatClass` type.

```
FormatClass = (fPositive,fNegative,fZero);
```

The Script Run Status Record

The `FindScriptRun` function returns the script run status record, defined by the `ScriptRunStatus` data type, when it completes its processing, which is to find a run of subscript text in a string. The `FindScriptRun` function is described in the section "FindScriptRun," which begins on page 5-81.

```
ScriptRunStatus =
RECORD
   script: SignedByte;  {script code of block}
   variant: SignedByte; {additional CharacterType information}
END;
```

Field descriptions

`script`	The script code of the subscript run. Zero indicates the Roman script system.
`variant`	Script-specific information about the run, in the same format as that returned by the `CharacterType` function, described in the chapter "Script Manager" in this book. This information includes the type of subscript—for example, Kanji, Katakana, or Hiragana for a Japanese script system.

Routines

This section describes the routines that you use to work with strings in your application, including sorting strings, modifying the contents of strings, converting dates and times to and from strings, and converting numbers to and from strings.

Defining and Specifying Strings

This section describes two routines that you can use to work with string handles and two routines for accessing string resources.

- The `NewString` function creates a copy of the specified string as a relocatable object in the heap.
- The `SetString` procedure changes the contents of a string that has already been allocated in the heap.
- The `GetString` function loads a string from a resource of type `'STR '`, reading the string from the resource file if necessary.
- The `GetIndString` procedure copies a string from a string list that is contained in a resource of type `'STR#'`.

NewString

The `NewString` function allocates memory in the heap for a string, copies its contents, and produces a handle for the heap version of the string.

```
FUNCTION NewString (theString: Str255): StringHandle;
```

`theString` A Pascal string that you want copied onto the heap.

DESCRIPTION

`NewString` returns a handle to the newly allocated string. If the string cannot be allocated, `NewString` returns `NIL`. The size of the allocated string is based on the actual length of `theString`, which may not be 255 bytes.

Note
Before using Pascal string functions that can change the length of the string, it is a good idea to maximize the size of the string object on the heap. You can call either the `SetString` procedure or the Memory Manager procedure `SetHandleSize` to modify the string's size. ◆

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `NewString` function is

Trap macro

`_NewString`

SetString

The `SetString` procedure changes the contents of a string referenced by a string handle, replacing the previous contents by copying the specified string.

```
PROCEDURE SetString (h: StringHandle; theString: Str255);
```

`h`	A handle to the string in memory whose contents you are replacing.
`theString`	A Pascal string.

DESCRIPTION

The `SetString` procedure sets the string whose handle is passed in the `h` parameter to the string specified by the parameter `theString`. If the new string is larger than the string originally referenced by `h`, `SetString` automatically resizes the handle and copies in the contents of the specified string.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `SetString` procedure is

Trap macro

`_SetString`

GetString

The `GetString` function loads a string from a string (`'STR '`) resource into memory. It returns a handle to the string with the specified resource ID, reading it from the resource file if necessary.

```
FUNCTION GetString (stringID: Integer): StringHandle;
```

`stringID`	The resource ID of the string (`'STR '`) resource containing the string.

DESCRIPTION

The `GetString` function returns a handle to a string with the specified resource ID. If `GetString` cannot read the resource, it returns `NIL`.

`GetString` calls the `GetResource` function of the Resource Manager to access the string. This means that if the specified resource is already in memory, `GetString` simply returns its handle.

Like the `NewString` function, `GetString` returns a handle whose size is based upon the actual length of the string.

Note
If your application uses a large number of strings, it is more efficient to store them in a string list (`'STR#'`) resource than as individual resources in the resource file. You then use the `GetIndString` procedure to access each string in the list. ◆

SPECIAL CONSIDERATIONS

`GetString` does not create a copy of the string.

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `GetString` function is

Trap macro

`_GetString`

GetIndString

The `GetIndString` procedure loads a string from a string list (`'STR#'`) resource into memory. It accesses the string by using the resource ID of the string list and the index of the individual string in that list. The list is read from the resource file if necessary.

```
PROCEDURE GetIndString (VAR theString: Str255; strListID: Integer;
                       index: Integer);
```

`theString`	On output, the Pascal string result.
`strListID`	The resource ID of the `'STR#'` resource that contains the string list.
`index`	The index of the string in the list. This is a value from 1 to the number of strings in the list that is referenced by the `strListID` parameter.

DESCRIPTION

`GetIndString` returns in the parameter `theString` a copy of the string from a string list that has the resource ID provided in the `strListID` parameter. If the resource that you specify cannot be read or the index that you specify is out of range for the string list, `GetIndString` sets `theString` to an empty string.

If necessary, `GetIndString` reads the string list from the resource file by calling the Resource Manager function `GetResource`. `GetIndString` accesses the string specified by the `index` parameter and copies it into `theString`. The index can range from 1 to the number of strings in the list.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

There is no trap macro for the `GetIndString` procedure. Instead, you need to use the `_GetResource` trap macro with the resource type (`'STR#'`) and string index.

Comparing Strings for Equality

This section describes text routines that you can use to determine whether two strings are equal.

Some of the routines operate on Pascal strings and others on text strings. Pascal strings are stored using standard Pascal string representation, which precedes the text characters with a length byte; these strings are limited to 255 bytes of data. Text strings do not use a length byte and can be up to 32 KB in length. Pascal strings are passed directly as parameters, while text strings are specified by an address value and an integer length value.

- The `EqualString` function compares two Pascal strings using the comparison rules of the Macintosh file system. This function does not make use of any script or language information.
- The `IdenticalString` function compares two Pascal strings for equality, making use of the string comparison information from a specified resource.
- The `IdenticalText` function compares two text strings for equality, making use of the string comparison information from a specified resource.

EqualString

The `EqualString` function compares two Pascal strings for equality, using the comparison rules of the Macintosh file system. The comparison performed by `EqualString` is a simple, character-by-character value comparison. This function does not make use of any script or language information; it assumes the use of a Roman script system.

```
FUNCTION EqualString (aStr, bStr: Str255;
                      caseSens, diacSens: Boolean): Boolean;
```

`aStr`	One of the Pascal strings to be compared.
`bStr`	The other Pascal string to be compared.
`caseSens`	A flag that indicates how to handle case-sensitive information during the comparison. If the value of `caseSens` is `TRUE`, uppercase characters are distinguished from the corresponding lowercase characters. If it is `FALSE`, case information is ignored.
`diacSens`	A flag that indicates how to handle information about diacritical marks during the string comparison. If the value of `diacSens` is `TRUE`, characters with diacritical marks are distinguished from the corresponding characters without diacritical marks during the comparison. If it is `FALSE`, diacritical marks are ignored.

DESCRIPTION

`EqualString` returns `TRUE` if the two strings are equal and `FALSE` if they are not equal. If its value is `TRUE`, `EqualString` distinguishes uppercase characters from the corresponding lowercase characters. If its value is `FALSE`, EqualString ignores diacritical marks during the comparison.

SPECIAL CONSIDERATIONS

The `EqualString` function is not localizable.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `EqualString` function are

Trap macro	Selector
`_CmpString`	$A03C

This trap macro can take optional arguments, each of which changes the default setting used by the macro when it is called without arguments. Each of these arguments corresponds to the Boolean parameters that are used with the Pascal function call. The various permutations of this trap macro are shown below; you must type each exactly as it is shown. The syntax shown here applies to the MPW Assembler; if you are using another development system, be sure to consult its documentation for the proper syntax.

Macro permutation	Value of diacSens	Value of caseSens
`_CmpString`	FALSE	FALSE
`_CmpString ,MARKS`	TRUE	FALSE
`_CmpString ,CASE`	FALSE	TRUE
`_CmpString ,MARKS,CASE`	TRUE	TRUE

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of the first string
A1	pointer to first character of the second string
D0	high-order word: number of bytes in the first string low-order word: number of bytes in the second string

Registers on exit

D0	long word result: 0 if strings are equal, 1 if strings are not equal

IdenticalString

The `IdenticalString` function compares two Pascal strings for equality, making use of the string comparison information from a resource that you specify as a parameter. `IdenticalString` uses only primary differences in its comparison.

```
FUNCTION IdenticalString (aStr, bStr: Str255;
                          itl2Handle: Handle): Integer;
```

`aStr`	One of the Pascal strings to be compared.
`bStr`	The other Pascal string to be compared.

`itl2Handle`	A handle to a string-manipulation (`'itl2'`) resource that contains string comparison information.

DESCRIPTION

`IdenticalString` returns 0 if the two strings are equal and 1 if they are not equal. It compares the two strings without regard for secondary sorting order, the meaning of which depends on the language of the strings. For example, for the English language, using only primary differences means that `IdenticalString` ignores diacritical marks and does not distinguish between lowercase and uppercase. For example, if the two strings are `'Rose'` and `'rosé'`, `IdenticalString` considers them equal and returns 0.

The `itl2Handle` parameter is used to specify a string-manipulation resource. If the value of this parameter is `NIL`, `IdenticalString` makes use of the resource for the current script. The string-manipulation resource includes tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Specifying a resource as a parameter is described in the section "Obtaining Resource Information," which begins on page 5-4.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

There is no trap macro for the `IdenticalString` function. Instead, you must convert the Pascal string into a text string by creating a pointer to its first character and finding its length, and then use the same macro as you do for the `IdenticalText` function, which is described next.

IdenticalText

The `IdenticalText` function compares two text strings for equality, making use of the string comparison information from a resource that you specify as a parameter. `IdenticalText` uses only primary sorting order in its comparison.

```
FUNCTION IdenticalText (aPtr, bPtr: Ptr; aLen, bLen: Integer;
                           itl2Handle: Handle): Integer;
```

`aPtr`	A pointer to the first character of the first text string.
`bPtr`	A pointer to the first character of the second text string.
`aLen`	The number of bytes in the first text string.

`bLen`	The number of bytes in the second text string.
`itl2Handle`	A handle to a string-manipulation (`'itl2'`) resource that contains string comparison information.

DESCRIPTION

`IdenticalText` returns 0 if the two text strings are equal and 1 if they are not equal. It compares the strings without regard for secondary sorting order, which means that it ignores diacritical marks and does not distinguish between lowercase and uppercase. For example, if the two text strings are `'Rose'` and `'rosé'`, `IdenticalText` considers them equal and returns 0.

The `itl2Handle` parameter is used to specify a string-manipulation resource. If the value of this parameter is `NIL`, `IdenticalText` makes use of the resource for the current script. The string-manipulation resource includes routines and tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Specifying a resource as a parameter is described in the section "Obtaining Resource Information," which begins on page 5-4.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `IdenticalText` function are

Trap macro	Selector
`_Pack6`	$001C

Determining Sorting Order for Strings in Different Languages

This section describes the Text Utilities routines available to help you sort strings in different languages. When strings from different languages occur in a single list, you must separate the strings so that all strings from each script system are contained in their own list. You then sort the list for each script system, using the sorting rules for that language and script system. You can then concatenate the individual language lists together, ordering the lists according to language and script ordering information that is found in the international script-sorting (`'itlm'`) resource.

- The `ScriptOrder` function indicates the order in which text from two different script systems should be sorted.
- The `LanguageOrder` function indicates the order in which text from two different languages from the same script system should be sorted.

- The `StringOrder` function determines the appropriate sorting order for two Pascal strings, taking into account the script and language codes of each.
- The `TextOrder` function determines the appropriate sorting order for two text strings, taking into account the script and language codes of each.

Note
When determining the order in which text from two different script systems should be sorted, the system script always sorts first, and scripts that are not enabled and installed always sort last. Invalid script or language codes always sort after valid ones. ◆

Script systems and the enabling and installing of scripts are described in the chapter "Script Manager" in this book and the script-sorting resource is described in the appendix "International Resources" in this book.

Pascal strings are stored using standard Pascal string representation, which precedes the text characters with a length byte; these strings are limited to 255 characters. Text strings do not use a length byte and can be up to 32 KB in length. Pascal strings are passed directly as parameters, while text strings are specified by two parameters: an address value and an integer length value.

The functions `LanguageOrder`, `StringOrder`, and `TextOrder` accept as parameters the implicit language codes listed in Table 5-13, as well as the explicit language codes listed in the chapter "Script Manager."

Table 5-13 Implicit language codes

Constant	Value	Explanation
`systemCurLang`	–2	Current language for system script (from `'itlb'`)
`systemDefLang`	–3	Default language for system script (from `'itlm'`)
`currentCurLang`	–4	Current language for current script (from `'itlb'`)
`currentDefLang`	–5	Default language for current script (from `'itlm'`)
`scriptCurLang`	–6	Current language for specified script (from `'itlb'`)
`scriptDefLang`	–7	Default language for specified script (from `'itlm'`)

ScriptOrder

The `ScriptOrder` function determines the order in which strings in two different scripts should be sorted.

```
FUNCTION ScriptOrder (script1, script2: ScriptCode): Integer;
```

`script1`	The script code of the first script.
`script2`	The script code of the second script.

DESCRIPTION

`ScriptOrder` takes a pair of script codes and determines in which order strings from the first script system should be sorted relative to strings from the second script system. It returns a value that indicates the sorting order: –1 if strings in the first script should be sorted before strings in the second script are sorted, 1 if strings in the first script should be sorted after strings in the second script are sorted, or 0 if the sorting order does not matter (that is, if the scripts are the same).

The script code values are listed in the chapter "Script Manager" in this book.

Note
Text of the system script is always first in a sorted list, regardless of the result returned by this function. ◆

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ScriptOrder` function are

Trap macro	Selector
`_Pack6`	$001E

LanguageOrder

The `LanguageOrder` function determines the order in which strings in two different languages should be sorted.

```
FUNCTION LanguageOrder (language1, language2: LangCode): Integer;
```

`language1`	The language code of the first language.
`language2`	The language code of the second language.

DESCRIPTION

`LanguageOrder` takes a pair of language codes and determines in which order strings from the first language should be sorted relative to strings from the second language. It returns a value that indicates the sorting order: –1 if strings in the first language should be sorted before sorting text in the second language, 1 if strings in the first language should be sorted after sorting strings in the second language, or 0 if the sorting order does not matter (that is, if the languages are the same).

Explicit language code values are listed in the chapter "Script Manager"; implicit language codes are listed in Table 5-13 on page 5-55 of this chapter. The implicit language codes `scriptCurLang` and `scriptDefLang` are not valid for `LanguageOrder` because the script system being used is not specified as a parameter to this function.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LanguageOrder` function are

Trap macro	Selector
`_Pack6`	$0020

StringOrder

The `StringOrder` function compares two Pascal strings, taking into account the script system and language for each of the strings. It takes both primary and secondary sorting orders into consideration and returns a value that indicates whether the first string is less than, equal to, or greater than the second string.

```
FUNCTION StringOrder (aStr, bStr: Str255;
                      aScript, bScript: ScriptCode;
                      aLang, bLang: LangCode): Integer;
```

`aStr`	One of the Pascal strings to be compared.
`bStr`	The other Pascal string to be compared.
`aScript`	The script code for the first string.
`bScript`	The script code for the second string.
`aLang`	The language code for the first string.
`bLang`	The language code for the second string.

DESCRIPTION

`StringOrder` returns –1 if the first string is less than the second string, 0 if the first string is equal to the second string, and 1 if the first string is greater than the second string. The ordering of script and language codes, which is based on information in the script-sorting resource, is considered in determining the relationship of the two strings.

Script code values and explicit language code values are listed in the chapter "Script Manager"; implicit language codes are listed in Table 5-13 on page 5-55 of this chapter.

Most applications specify the language code `scriptCurLang` for both the `aLang` and `bLang` values.

`StringOrder` first calls `ScriptOrder`; if the result of `ScriptOrder` is not 0 (that is, if the strings use different scripts), `StringOrder` returns the same result.

`StringOrder` next calls `LanguageOrder`; if the result of `LanguageOrder` is not 0 (that is, if the strings use different languages), `StringOrder` returns the same result.

At this point, `StringOrder` has two strings that are in the same script and language, so it compares them by using the sorting rules for that script and language, applying both the primary and secondary sorting orders. If that script is not installed and enabled (as described in the chapter "Script Manager" in this book), it uses the sorting rules specified by the system script or the font script, depending on the state of the international resources selection flag. See the section "Obtaining Resource Information," beginning on page 5-4.

The `StringOrder` function is primarily used to insert Pascal strings in a sorted list; for sorting, rather than using this function, it may be faster to sort first by script and language by using the `ScriptOrder` and `LanguageOrder` functions, and then to call the `CompareString` function, described on page 5-62, to sort strings within a script or language group.

SPECIAL CONSIDERATIONS

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

TextOrder

The `TextOrder` function compares two text strings, taking into account the script and language for each of the strings. It takes both primary and secondary sorting orders into consideration and returns a value that indicates whether the first string is less than, equal to, or greater than the second string.

```
FUNCTION TextOrder (aPtr, bPtr: Ptr; aLen, bLen: Integer;
                    aScript, bScript: ScriptCode;
                    aLang, bLang: LangCode): Integer;
```

`aPtr`	A pointer to the first character of the first text string.
`bPtr`	A pointer to the first character of the second text string.
`aLen`	The number of bytes in the first text string.
`bLen`	The number of bytes in the second text string.
`aScript`	The script code for the first text string.
`bScript`	The script code for the second text string.
`aLang`	The language code for the first text string.

`bLang`	The language code for the second text string.

DESCRIPTION

`TextOrder` returns –1 if the first string is less than the second string, 0 if the first string is equal to the second string, and 1 if the first string is greater than the second string. The ordering of script and language codes, which is based on information in the script-sorting resource, is considered in determining the relationship of the two strings.

Script code values and explicit language code values are listed in the chapter "Script Manager"; implicit language codes are listed in Table 5-13 on page 5-55 of this chapter. Most applications specify the language code `scriptCurLang` for both the `aLang` and `bLang` values.

`TextOrder` first calls `ScriptOrder`; if the result of `ScriptOrder` is not 0 (that is, if the strings use different scripts), `TextOrder` returns the same result.

`TextOrder` next calls `LanguageOrder`; if the result of `LanguageOrder` is not 0 (that is, if the strings use different languages), `TextOrder` returns the same result.

At this point, `TextOrder` has two strings that are in the same script and language, so it compares them by using the sorting rules for that script and language, applying both the primary and secondary sorting orders. If that script is not installed and enabled (as described in the chapter "Script Manager" in this book), it uses the sorting rules specified by the system script or the font script, depending on the state of the international resources selection flag. See the section "Obtaining Resource Information," beginning on page 5-4.

The `TextOrder` function is primarily used to insert text strings in a sorted list; for sorting, rather than using this function, it may be faster to sort first by script and language by using the `ScriptOrder` and `LanguageOrder` functions, and then to call the `CompareText` function, described on page 5-63, to sort strings within a script or language group.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TextOrder` function are

Trap macro	Selector
`_Pack6`	$0022

Determining Sorting Order for Strings in the Same Language

This section describes text routines that you can use to determine the sorting order of two strings.

Some of the routines operate on Pascal strings and others on text strings. Pascal strings are stored using standard Pascal string representation, which precedes the text characters with a length byte; these strings are limited to 255 bytes of data. Text strings do not use a length byte and can be up to 32 KB in length. Pascal strings are passed directly as parameters, while text strings are specified by an address value and an integer length value.

- The `RelString` function compares two Pascal strings using the string comparison rules of the Macintosh file system. This function does not make use of any script or language information, assuming the use of a Roman script system.
- The `CompareString` function compares two Pascal strings, making use of the string comparison information from a specified resource.
- The `CompareText` function compares two text strings for equality, making use of the string comparison information from a specified resource.

RelString

The `RelString` function compares two Pascal strings using the string comparison rules of the Macintosh file system and returns a value that indicates the sorting order of the first string relative to the second string. This function does not make use of any script or language information; it assumes the original Macintosh character set only. `RelString` uses the sorting rules that are described in Table 5-2 on page 5-17.

```
FUNCTION RelString (aStr, bStr: Str255;
                    caseSens, diacSens: Boolean): Integer;
```

`aStr`	One of the Pascal strings to be compared.
`bStr`	The other Pascal string to be compared.
`caseSens`	A flag that indicates how to handle case-sensitive information during the comparison. If the value of `caseSens` is `TRUE`, uppercase characters are distinguished from the corresponding lowercase characters. If it is `FALSE`, case information is ignored.
`diacSens`	A flag that indicates how to handle information about diacritical marks during the string comparison. If the value of `diacSens` is `TRUE`, characters with diacritical marks are distinguished from the corresponding characters without diacritical marks during the comparison. If it is `FALSE`, diacritical marks are ignored.

DESCRIPTION

`RelString` returns –1 if the first string is less than the second string, 0 if the two strings are equal, and 1 if the first string is greater than the second string. It compares the two strings in the same manner as does the `EqualString` function, by simply looking at the ASCII values of their characters. However, `RelString` provides more information about the two strings—it indicates their relationship to each other, rather than determining if they are exactly equal.

If the value of the `diacSens` parameter is `FALSE`, `RelString` ignores diacritical marks and strips them as shown in the appendix “International Resources” in this book.

If the value of the `caseSens` parameter is `FALSE`, the comparison is not case-sensitive; `RelString` performs a conversion from lowercase to uppercase characters.

SPECIAL CONSIDERATIONS

The `RelString` function is not localizable and does not work properly with non-Roman script systems.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `RelString` function are

Trap macro	Selector
`_RelString`	$A050

The trap macro for the `RelString` function can take optional arguments, each of which changes the default setting used by the macro when it is called without arguments. Each of these arguments corresponds to the Boolean parameters that are used with the Pascal function call. The various permutations of this trap macro are shown below; you must type each exactly as it is shown. The syntax shown here applies to the MPW Assembler; if you are using another development system, be sure to consult its documentation for the proper syntax.

Macro permutation		Value of `diacSens`	Value of `caseSens`
`_RelString`		`FALSE`	`FALSE`
`_RelString`	`,MARKS`	`TRUE`	`FALSE`
`_RelString`	`,CASE`	`FALSE`	`TRUE`
`_RelString`	`,MARKS,CASE`	`TRUE`	`TRUE`

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of the first string
A1	pointer to first character of the second string
D0	high-order word: number of bytes in the first string low-order word: number of bytes in the second string

Registers on exit

D0	long word result: –1 if first string is less than second, 0 if equal, 1 if first string is greater than second

CompareString

The `CompareString` function compares two Pascal strings, making use of the string comparison information from a resource that you specify as a parameter. It takes both primary and secondary sorting orders into consideration and returns a value that indicates the sorting order of the first string relative to the second string.

```
FUNCTION CompareString(aStr, bStr: Str255;
                       itl2Handle: Handle): Integer;
```

`aStr`	One of the Pascal strings to be compared.
`bStr`	The other Pascal string to be compared.
`itl2Handle`	The handle to the string-manipulation resource that contains string comparison information.

DESCRIPTION

`CompareString` returns –1 if the first string is less than the second string, 0 if the first string is equal to the second string, and 1 if the first string is greater than the second string.

The `itl2Handle` parameter is used to specify a string-manipulation resource. If the value of this parameter is `NIL`, `CompareString` makes use of the resource for the current script. The string-manipulation resource includes routines and tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Specifying a resource as a parameter is described in the section “Obtaining Resource Information,” beginning on page 5-4.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

There is no trap macro for the `CompareString` function. Instead, you must convert the Pascal string into a text string by creating a pointer to its first character and finding its length, and then use the same macro as you do for the `CompareText` function, which is described next.

CompareText

The `CompareText` function compares two text strings, making use of the string comparison information from a resource that you specify as a parameter. It takes both primary and secondary sorting orders into consideration and returns a value that indicates the sorting order of the first string relative to the second string.

```
FUNCTION CompareText (aPtr, bPtr: Ptr; aLen, bLen: Integer;
                      itl2Handle: Handle): Integer;
```

`aPtr`	A pointer to the first character of the first text string.
`bPtr`	A pointer to the first character of the second text string.
`aLen`	The number of bytes in the first text string.
`bLen`	The number of bytes in the second text string.
`itl2Handle`	A handle to a string-manipulation (`'itl2'`) resource that contains string comparison information.

DESCRIPTION

`CompareText` returns –1 if the first string is less than the second string, 0 if the first string is equal to the second string, and 1 if the first string is greater than the second string.

The `itl2Handle` parameter is used to specify a string-manipulation resource. If the value of this parameter is `NIL`, `CompareText` makes use of the resource for the current script. The string-manipulation resource includes routines and tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Specifying a resource as a parameter is described in the section "Obtaining Resource Information," beginning on page 5-4.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `CompareText` function are

Trap macro	Selector
`_Pack6`	$001A

Modifying Characters and Diacritical Marks

This section provides details on text routines that you can use to modify the characters in text:

- The `UpperString` procedure converts any lowercase letters in a Pascal string to their uppercase equivalents. `UpperString` uses the Macintosh file system string-manipulation rules, which means that it only works properly for Roman characters with codes through $D8.

The following four routines use tables in the string-manipulation (`'itl2'`) resource to perform their character-mapping operations. This allows you to customize their operation for different countries.

- The `LowercaseText` procedure converts any uppercase characters in a text string into their lowercase equivalents, making use of the conversion rules for the specified script system.
- The `UppercaseText` procedure converts any lowercase characters in a text string into their uppercase equivalents, making use of the conversion rules for the specified script system.
- The `StripDiacritics` procedure strips diacritical characters from a text string, making use of the conversion rules for the specified script system.
- The `UppercaseStripDiacritics` procedure strips diacritical marks and converts lowercase characters into their uppercase equivalents in a text string, making use of the conversion rules for the specified script system.

UpperString

The `UpperString` procedure converts any lowercase letters in a Pascal string to their uppercase equivalents. This procedure converts characters using the Macintosh file system rules, which means that only a subset of the Roman character set (character codes with values through $D8) are converted. These rules are summarized in Table 5-2 on page 5-17. Use this procedure to emulate the behavior of the Macintosh file system.

```
PROCEDURE UpperString (VAR theString: Str255; diacSens: Boolean);
```

`theString`	On input, this is the Pascal string to be converted. On output, this contains the string resulting from the conversion.
`diacSens`	A flag that indicates whether the case conversion is to strip diacritical marks. If the value of this parameter is `FALSE`, diacritical marks are stripped.

DESCRIPTION

`UpperString` traverses the characters in `theString` and converts any lowercase characters with character codes in the range $0 through $D8 into their uppercase equivalents. If the `diacSens` flag is `TRUE`, diacritical marks are considered in the conversion; if it is `FALSE`, any diacritical marks are stripped. `UpperString` places the converted characters in `theString`.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `UpperString` procedure are

Trap macro	Selector
`_UprString`	$A054

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of string
D0	the length of the string (a word value)

Registers on exit

A0	pointer to first character of string

The trap macro for the `UpperString` procedure can take an optional argument, which changes the default setting used by the macro when it is called without arguments. This argument corresponds to the Boolean parameter `diacSens` that is used with the Pascal function call. The permutations of this trap macro are shown below; you must type each exactly as it is shown.

The syntax shown here applies to the MPW Assembler; if you are using another development system, be sure to consult its documentation for the proper syntax.

Macro permutation	Value of `diacSens`
`_UprString`	`TRUE`
`_UprString ,MARKS`	`FALSE`

LowercaseText

The `LowercaseText` procedure converts any uppercase characters in a text string into their lowercase equivalents. The text string can be up to 32 KB in length.

```
PROCEDURE LowercaseText (textPtr: Ptr; len: Integer;
                         script: ScriptCode);
```

`textPtr`	A pointer to the text string to be converted.
`len`	The number of bytes in the text string.
`script`	The script code for the script system whose resources are used to determine the results of converting characters.

DESCRIPTION

`LowercaseText` traverses the characters starting at the address specified by `textPtr` and continues for the number of characters specified in `len`. It converts any uppercase characters in the text into lowercase.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. The possible values for script codes are listed in the chapter "Script Manager" of this book. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

If `LowercaseText` cannot access the specified resource, it generates an error code and does not modify the string. You need to call the `ResError` function to determine which, if any, error occurred. `ResError` is described in the Resource Manager chapter of the book *Inside Macintosh: More Macintosh Toolbox*.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `LowercaseText` procedure is

Trap macro

`_LowerText`

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of string
D0	length of string in bytes (word); must be less than 32 KB

Registers on exit

D0	result code

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Can't get correct `'itl2'` resource or resource is not in current format

UppercaseText

The `UppercaseText` procedure converts any lowercase characters in a text string into their uppercase equivalents. The text string can be up to 32 KB in length.

```
PROCEDURE UppercaseText (textPtr: Ptr; len: Integer;
                         script: ScriptCode);
```

`textPtr`	A pointer to the text string to be converted.
`len`	The number of bytes in the text string.
`script`	The script code of the script system whose case conversion rules are used for determining uppercase character equivalents.

DESCRIPTION

`UppercaseText` traverses the characters starting at the address specified by `textPtr` and continues for the number of characters specified in `len`. It converts any lowercase characters in the text into uppercase.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. The possible values for script codes are listed in the chapter "Script Manager" of this book. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

If `UppercaseText` cannot access the specified resource, it generates an error code and does not modify the string. You need to call the `ResError` function to determine which, if any, error occurred. `ResError` is described in the Resource Manager chapter of the book *Inside Macintosh: More Macintosh Toolbox*.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `UppercaseText` procedure is

Trap macro

`_UpperText`

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of string
D0	length of string in bytes (word); must be less than 32 KB

Registers on exit

D0	result code

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Can't get correct `'itl2'` resource or resource is not in current format

StripDiacritics

The `StripDiacritics` procedure strips any diacritical marks from a text string. The text string can be up to 32 KB in length.

```
PROCEDURE StripDiacritics (textPtr: Ptr; len: Integer;
                           script: ScriptCode);
```

`textPtr`	A pointer to the text string to be stripped.
`len`	The number of bytes in the text string.
`script`	The script code for the script system whose rules are used for determining which character results from stripping a diacritical mark.

DESCRIPTION

`StripDiacritics` traverses the characters starting at the address specified by `textPtr` and continues for the number of characters specified in `len`. It strips any diacritical marks from the text.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. The possible values for script codes are listed in the chapter "Script Manager" of this book. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

If `StripDiacritics` cannot access the specified resource, it generates an error code and does not modify the string. You need to call the `ResError` function to determine which, if any, error occurred. `ResError` is described in the Resource Manager chapter of the book *Inside Macintosh: More Macintosh Toolbox*.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `StripDiacritics` procedure is

Trap macro

`_StripText`

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of string
D0	length of string in bytes (word); must be less than 32 KB

Registers on exit

D0	result code

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Can't get correct `'itl2'` resource or resource is not in current format

UppercaseStripDiacritics

The `UppercaseStripDiacritics` procedure converts any lowercase characters in a text string into their uppercase equivalents and strips any diacritical marks from the text. The text string can be up to 32 KB in length.

```
PROCEDURE UppercaseStripDiacritics (textPtr: Ptr; len: Integer;
                                    script: ScriptCode);
```

`textPtr`	A pointer to the text string to be converted.
`len`	The number of bytes in the text string.
`script`	The script code of the script system whose resources are used to determine the results of converting characters.

DESCRIPTION

`UppercaseStripDiacritics` traverses the characters starting at the address specified by `textPtr` and continues for the number of characters specified in `len`. It converts lowercase characters in the text into their uppercase equivalents and also strips diacritical marks from the text string. This procedure combines the effects of the `UppercaseText` and `StripDiacritics` procedures.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. The possible values for script codes are listed in the chapter “Script Manager” of this book. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

If `UppercaseStripDiacritics` cannot access the specified resource, it generates an error code and does not modify the string. You need to call the `ResError` function to determine which, if any, error occurred. `ResError` is described in the Resource Manager chapter of the book *Inside Macintosh: More Macintosh Toolbox*.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `UppercaseStripDiacritics` procedure is

Trap macro

`_StripUpperText`

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to first character of string
D0	length of string in bytes (word); must be less than 32 KB

Registers on exit

D0	result code

RESULT CODES

`noErr`	0	No error
`resNotFound`	–192	Can’t get correct `'itl2'` resource or resource is not in current format

Truncating Strings

This section describes two Text Utilities functions that you can use to truncate portions of strings. Each of these function can truncate characters from different locations in a string, and each makes use of the current script and font to perform its operation. The current script is defined on page 5-4. The current font is the font that is currently in use in the current graphics port.

- The `TruncString` function ensures that a Pascal string fits into the specified pixel width, by truncating the string as necessary.
- The `TruncText` function ensures that a text string fits into the specified pixel width, by truncating the string as necessary.

TruncString

The `TruncString` function ensures that a Pascal string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font.

```
FUNCTION TruncString (width: Integer; VAR theString: Str255;
                      truncWhere: TruncCode): Integer;
```

`width`	The number of pixels in which the string must be displayed in the current script and font.
`theString`	The Pascal string to be displayed. On output, contains a version of the string that has been truncated (if necessary) to fit in the number of pixels specified by `width`.
`truncWhere`	A constant that indicates where the string should be truncated. You must set this parameter to one of the constants `truncEnd` or `truncMiddle`.

DESCRIPTION

The `TruncString` function ensures that a Pascal string fits into the pixel width specified by the `width` parameter by modifying the string, if necessary, through truncation. `TruncString` uses the font script to determine how to perform truncation. If truncation occurs, `TruncString` inserts a truncation indicator, which is the ellipsis (…) in the Roman script system. You can specify which token to use for indicating truncation as the `tokenEllipsis` token type in the untoken table of a tokens (`'itl4'`) resource.

The `truncWhere` parameter specifies where truncation is performed. If you supply the `truncEnd` value, characters are truncated off the end of the string. If you supply the `truncMiddle` value, characters are truncated from the middle of the string; this is useful when displaying pathnames.

The `TruncString` function returns a result code that indicates whether the string was truncated.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TruncString` function are

Trap macro	Selector
`_ScriptUtil`	$8208 FFE0

RESULT CODES

`Truncated`	1	Truncation performed
`NotTruncated`	0	No truncation necessary
`TruncErr`	–1	Truncation necessary, but cannot be performed within the specified width
`resNotFound`	–192	Cannot get the correct `'itl4'` resource or resource is not in current format

SEE ALSO

To determine the width of a string in the current font and script, use the QuickDraw `StringWidth` function, which is described in the chapter "QuickDraw Text" in this book.

TruncText

The `TruncText` function ensures that a text string fits into the specified pixel width, by truncating the string as necessary. This function makes use of the current script and font. The text string can be up to 32 KB long.

You can use the `TruncText` function to ensure that a string defined by a pointer and a byte length fits into the specified pixel width, by truncating the string in a manner dependent on the font script.

```
FUNCTION TruncText (width: Integer; textPtr: Ptr;
                    VAR length: Integer;
                    truncWhere: TruncCode): Integer;
```

`width`	The number of pixels in which the text string must be displayed in the current script and font.
`textPtr`	A pointer to the text string to be truncated.
`length`	On input, the number of bytes in the text string to be truncated. On output, this value is updated to reflect the length of the (possibly) truncated text.
`truncWhere`	A constant that indicates where the text string should be truncated. You must set this parameter to one of the constants `truncEnd` or `truncMiddle`.

DESCRIPTION

The `TruncText` function ensures that a text string fits into the pixel width specified by the `width` parameter by modifying the string, if necessary, through truncation. `TruncText` uses the font script to determine how to perform truncation. If truncation occurs, `TruncText` inserts a truncation indicator which is the ellipsis (…) in the Roman script system. You can specify which token to use for indicating truncation as the `tokenEllipsis` token type in the untoken table of a tokens resource.

The `truncWhere` parameter specifies where truncation is performed. If you supply the `truncEnd` value, characters are truncated off the end of the string. If you supply the `truncMiddle` value, characters are truncated from the middle of the string; this is useful when displaying pathnames.

The `TruncText` function returns a result code that indicates whether the string was truncated.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TruncText` function are

Trap macro	Selector
`_ScriptUtil`	$820C FFDE

RESULT CODES

`Truncated`	1	Truncation performed
`NotTruncated`	0	No truncation necessary
`TruncErr`	–1	Truncation necessary, but cannot be performed within the specified width
`resNotFound`	–192	Cannot get the correct `'itl4'` resource or resource is not in current format

SEE ALSO

To determine the width of a string in the current font and script, use the QuickDraw `StringWidth` function, which is described in the chapter "QuickDraw Text" in this book.

Searching for and Replacing Strings

This section describes two Text Utilities routines that you can use to search for and replace strings in larger strings:

- The `ReplaceText` function searches a text string and replaces all instances of a target string with another string. `ReplaceText` uses the string-manipulation resource tables and works properly for all script systems, including 2-byte script systems.
- The `Munger` function searches text for a specified target string and replaces it with another string. This function operates on a byte-by-byte basis; thus, it does not always work for 2-byte script systems.

ReplaceText

The `ReplaceText` function searches text, replacing all instances of a string in that text with another string. `ReplaceText` searches on a character-by-character basis (as opposed to byte-by-byte), so it works properly for all script systems.

```
FUNCTION ReplaceText (baseText, substitutionText: Handle;
                      key: Str15): Integer;
```

`baseText` A handle to the string in which `ReplaceText` is to substitute text.

`substitutionText`
A handle to the string that `ReplaceText` uses as substitute text.

`key` A Pascal string of less than 16 bytes that `ReplaceText` searches for.

DESCRIPTION

`ReplaceText` searches the text specified by the `baseText` parameter for instances of the string in the `key` parameter and replaces each instance with the text specified by the `substitutionText` parameter. `ReplaceText` searches on a character-by-character basis. It recognizes 2-byte characters in script systems that contain them and advances the search appropriately after encountering a 2-byte character.

`ReplaceText` returns an integer value. If the returned value is positive, it indicates the number of substitutions performed; if it is negative, it indicates an error. The constant `noErr` is returned if there was no error and no substitutions were performed.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ReplaceText` function are

Trap macro	Selector
`_ScriptUtil`	$820C FFDC

RESULT CODES

`nilHandleErr`	109	`GetHandleSize` fails on `baseText` or `substitutionText`
`memFullErr`	108	`SetHandleSize` fails on `baseText`
`memWZErr`	–111	`GetHandleSize` fails on `baseText` or `substitutionText`

Munger

The `Munger` function searches text for a specified string pattern and replaces it with another string.

```
FUNCTION Munger (h:  Handle; offset: LongInt; ptr1: Ptr;
                 len1: LongInt; ptr2: Ptr; len2: LongInt): LongInt;
```

`h`	A handle to the text string that is being manipulated.
`offset`	The byte offset in the destination string at which `Munger` begins its operation.
`ptr1`	A pointer to the first character in the string for which `Munger` is searching.
`len1`	The number of bytes in the string for which `Munger` is searching.
`ptr2`	A pointer to the first character in the substitution string.
`len2`	The number of bytes in the substitution string.

DESCRIPTION

`Munger` manipulates bytes in a string to which you specify a handle in the `h` parameter. The manipulation begins at a byte offset, specified in `offset`, in the string. Munger searches for the string specified by `ptr1` and `len1`; when it finds an instance of that string, it replaces it with the substitution string, which is specified by `ptr2` and `len2`.

IMPORTANT

`Munger` operates on a byte-by-byte basis, which can produce inappropriate results for 2-byte script systems. The `ReplaceText` function works properly for all languages. You are encouraged to use `ReplaceText` instead of `Munger` whenever possible. ▲

`Munger` takes special action if either of the specified pointer values is `NIL` or if either of the length values is `0`.

- If `ptr1` is `NIL`, `Munger` replaces characters without searching. It replaces `len1` characters starting at the `offset` location with the substitution string.
- If `ptr1` is `NIL` and `len1` is negative, `Munger` replaces all of the characters from the `offset` location to the end of the string with the substitution string.
- If `len1` is 0, `Munger` inserts the substitution string without replacing anything. `Munger` inserts the string at the `offset` location and returns the offset of the first byte past where the insertion occurred.
- If `ptr2` is `NIL`, `Munger` searches but does not replace. In this case, `Munger` returns the offset at which the string was found.
- If `len2` is 0 and `ptr2` is not `NIL`, `Munger` searches and deletes. In this case, `Munger` returns the offset at which it deleted.
- If the portion of the string from the `offset` location to its end matches the beginning of the string that `Munger` is searching for, `Munger` replaces that portion with the substitution string.

`Munger` returns a negative value when it cannot find the designated string.

▲ **WARNING**

Be careful not to specify an offset with a value that is greater than the length of the destination string. Unpredictable results may occur. ▲

SPECIAL CONSIDERATIONS

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

The destination string must be in a relocatable block that was allocated by the Memory Manager.

SEE ALSO

`Munger` calls the `GetHandleSize` and `SetHandleSize` routines to access or modify the length of the string it is manipulating. These routines are described in the book *Inside Macintosh: Memory Manager.*

ASSEMBLY-LANGUAGE INFORMATION

The trap macro for the `Munger` function is

Trap macro

`_Munger`

Working With Word, Script, and Line Boundaries

This section describes the text routines that you can use to edit and display formatted text. These functions all take into account script and language considerations, making use of tables in the string-manipulation (`'itl2'`) resource in their computations.

- The `FindWordBreaks` procedure determines the beginning and ending boundaries of a word in a text string.
- The `StyledLineBreak` function determines the proper location at which to break a line of text that may contain multiple script runs, breaking it on a word boundary if possible.
- The `FindScriptRun` function finds the next boundary between main text and a specified subscript within a script run.

FindWordBreaks

The `FindWordBreaks` procedure determines the beginning and ending boundaries of a word in a text string.

```
PROCEDURE FindWordBreaks(textPtr: Ptr; textLength: Integer;
                  offset: Integer; leadingEdge: Boolean;
                  nbreaks: BreakTablePtr;
                  VAR offsets: OffsetTable; script: ScriptCode);
```

`textPtr`	A pointer to the text string to be examined.
`textLength`	The number of bytes in the text string.
`offset`	A byte offset into the text. This parameter plus the `leadingEdge` parameter determine the position of the character at which to start the search.

`leadingEdge`	A flag that specifies which character should be used to start the search. If `leadingEdge` is `TRUE`, the search starts with the character specified in the `offset` parameter; if it is `FALSE`, the search starts with the character previous to the offset.
`nbreaks`	A pointer to a word-break table of type `NBreakTable` or `BreakTable`. If the value of this pointer is 0, the default word-break table of the script system specified by the `script` parameter is used. If the value of this pointer is –1, the default line-break table of the specified script system is used.
`offsets`	On output, the values in this table indicate the boundaries of the word that has been found.
`script`	The script code for the script system whose tables are used to determine where word boundaries occur.

DESCRIPTION

`FindWordBreaks` searches for a word in a text string. The `textPtr` and `textLength` parameters specify the text string that you want searched. The `offset` parameter and `leadingEdge` parameter together indicate where the search begins.

If `leadingEdge` is `TRUE`, the search starts at the character at the `offset`. If `leadingEdge` is `FALSE`, the search starts at the character preceding the `offset` position.

`FindWordBreaks` searches backward through the text string for one of the word boundaries and forward through the text string for its other boundary. It uses the definitions in the table specified by `nbreaks` to determine what constitutes the boundaries of a word. Each script system's word-break table is part of its string-manipulation (`'itl2'`) resource. The format of the `NBreakTable` record is described in the appendix "International Resources" in this book.

`FindWordBreaks` returns its results in an `OffsetTable` record, the format of which is described in the section "The Offset Table Record" on page 5-44. `FindWordBreaks` uses only the first element of this three-element table. Each element is a pair of integers: `offFirst` and `offSecond`.

`FindWordBreaks` places the offset from the beginning of the text string to just before the leading edge of the character of the word that it finds in the `offFirst` field.

`FindWordBreaks` places the offset from the beginning of the text string to just after the trailing edge of the last character of the word that it finds in the `offSecond` field. For example, if the text "This is it" is passed with `offset` set to 0 and `leadingEdge` set to `TRUE`, then `FindWordBreaks` returns the offset pair (0,4).

If `leadingEdge` is `TRUE` and the value of `offset` is 0, then `FindWordBreaks` returns the offset pair (0,0). If `leadingEdge` is `FALSE` and the value of `offset` equals the value of `textLength`, then `FindWordBreaks` returns the offset pair with values (`textLength`, `textLength`).

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `FindWordBreaks` procedure are

Trap macro	Selector
`_ScriptUtil`	$C012 001A

StyledLineBreak

The `StyledLineBreak` function returns the proper location to break a line of text. It breaks the line on a word boundary if possible and allows for multiscript runs and style runs on a single line.

```
FUNCTION StyledLineBreak(textPtr: Ptr; textLen: LongInt;
                  textStart, textEnd, flags: LongInt;
                  VAR textWidth: Fixed;
                  VAR textOffset: LongInt): StyledLineBreakCode;
```

`textPtr`	A pointer to the beginning of a script run on the current line to be broken.
`textLen`	The number of bytes in the script run on the current line to be broken.
`textStart`	A byte offset to the beginning of a style run within the script run.
`textEnd`	A byte offset to the end of the style run within the script run.
`flags`	Reserved for future expansion; must be 0.
`textWidth`	The maximum length of the displayed line in pixels. `StyledLineBreak` decrements this value for its own use. Your responsibility is to set it before your first call to `StyledLineBreak` for a line.
`textOffset`	Must be nonzero on your first call to `StyledLineBreak` for a line, and zero for subsequent calls to `StyledLineBreak` for that line. This value allows `StyledLineBreak` to differentiate between the first and subsequent calls, which is important when a long word is found (as described below). On output, `textOffset` is the count of bytes from `textPtr` to the location in the text string where the line break is to occur.

DESCRIPTION

Use the `StyledLineBreak` function when you are laying out lines in an environment that may include text from multiple scripts. To use this function, you need to understand how QuickDraw draws text, which is described in the chapter “QuickDraw Text” in this book.

You can only use the StyledLineBreak function when you have organized your text in script runs and style runs within each script run. This type of text organization is used by most text-processing applications that allow for multiscript text. Use this function when you are displaying text in a screen area to determine the best place to break each displayed line. For an overview of how to use this function, read the section "Finding Line Breaks" beginning on page 5-24.

What you do is iterate through your text, a script run at a time starting from the first character past the end of the previous line. Use StyledLineBreak to check each style run in the script run (in memory order) until the function determines that it has arrived at a line break. As you loop through each style run, before calling StyledLineBreak, you must set the text values in the graphics port record that are used by QuickDraw to measure the text. These include the font, font size, and font style of the style run. An example of a loop that uses this function is found in Listing 5-4 on page 5-27.

When used with unformatted text, textStart can be 0, and textEnd is identical to textLen. With styled text, the interval between textStart and textEnd specifies a style run. The interval between textPtr and textLen specifies a script run. Note that the style runs in StyledLineBreak must be traversed in memory order, not in display order.

If the current style run is included in a contiguous sequence of other style runs of the same script, then textPtr should point to the start of the first style run of the same script on the line, and textLen should include the last style run of the same script on the line. This is because word boundaries can extend across style runs, but not across script runs.

StyledLineBreak automatically decrements the textWidth variable by the width of the style run for use on the next call. You need to set the value of textWidth before calling it to process a line.

The textOffset parameter must be nonzero for the first call on a line and zero for each call to the function on the line. This allows StyledLineBreak to act differently when a long word is encountered: if the word is in the first style run on the line, StyledLineBreak breaks the line on a character boundary within the word; if the word is in a subsequent style run on the line, StyledLineBreak breaks the line before the start of the word.

When StyledLineBreak finds a line break, it sets the value of textOffset to the count of bytes that can be displayed starting at textPtr.

IMPORTANT
When StyledLineBreak is called for the second or subsequent style runs within a script run, the textOffset value at exit may be less than the textStart parameter (that is, it may specify a line break before the current style run). ▲

Although the offsets are in long integer values and the widths are in fixed values for future extensions, in the current version the long integer values are restricted to the integer range, and only the integer portion of the widths is used.

StyledLineBreak always chooses a line break for the last style run on the line in memory order as if all whitespace in that style run would be stripped. The VisibleLength function, which is a QuickDraw function used to eliminate trailing spaces from a style run before drawing it, can be called for the style run that is at the display end of a line. This leads to a potential conflict when both functions are used with mixed-directional text: if the end of a line in memory order actually occurs in the middle of the displayed line, StyledLineBreak assumes that the whitespace is stripped from that run, but VisibleLength does not strip the characters. The VisibleLength function is described in the chapter "QuickDraw Text" in this book.

The StyledLineBreak result (defined by the StyledLineBreakCode data type) indicates whether the function broke on a word boundary or a character boundary, or if the width extended beyond the edge of the text.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the StyledLineBreak function are

Trap macro	Selector
_ScriptUtil	$821C FFFE

RESULT CODES

BreakOverflow	2	No line break is yet necessary, since the current style run still fits on the line
BreakChar	1	Line breaks on character boundary
BreakWord	0	Line breaks on word boundary

SEE ALSO

For details on the VisibleLength, TextWidth, and PortionText functions, see the chapter "QuickDraw Text" in this book.

FindScriptRun

The FindScriptRun function finds the next block of subscript text within a script run.

```
FUNCTION FindScriptRun (textPtr: Ptr;
                        textLen: LongInt;
                        VAR lenUsed: LongInt): ScriptRunStatus;
```

textPtr	A pointer to the text string to be analyzed.
textLen	The number of bytes in the text string.
lenUsed	On output, contains the length, in bytes, of the script run that begins with the first character in the string; this length is always greater than or equal to 1, unless the string passed in is of length 0.

DESCRIPTION

The `FindScriptRun` function is used to identify blocks of subscript text in a string. Some script systems include subscripts, which are character sets that are subsidiary to the main character set. One useful subscript is the set of all character codes that have the same meaning in Roman as they do in a non-Roman script. For other scripts such as Japanese, there are additional useful subscripts. For example, a Japanese script system might include some Hiragana characters that are useful for input methods.

`FindScriptRun` computes the length of the current run of subscript text in the text string specified by `textPtr` and `textLen`. It assigns the length, in bytes, to the `lenUsed` parameter and returns a status code. You can advance the text pointer by the value of `lenUsed` to make subsequent calls to this function. You can use this function to identify runs of subscript characters so that you can treat them separately.

The function result identifies the run as either native text, Roman, or one of the defined subscripts of the script system and returns a record of type `ScriptRunStatus`. This record is described in the section "The Script Run Status Record" on page 5-46.

Word processors and other applications can call `FindScriptRun` to separate style runs of native text from non-native text. You can use this capability to extract those characters and apply a different font to them. Figure 5-11 on page 5-28 provides an example of using the `FindScriptRun` routine.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `FindScriptRun` function are

Trap macro	Selector
`_ScriptUtil`	$820C 0026

Converting Date and Time Strings Into Numeric Representations

This section describes the Text Utilities routines that you can use to convert date and time strings into numeric representations:

- The `InitDateCache` function initializes the date cache record, which is used by the `StringToDate` and `StringToTime` functions.
- The `StringToDate` function parses text for a date specification and fills in numeric date information in a `LongDateRec` record.
- The `StringToTime` function parses text for a time specification and fills in numeric time information in a `LongDateRec` record.

InitDateCache

The `InitDateCache` function initializes the date cache record, which is used to store data for use by the `StringToDate` and `StringToTime` functions.

```
FUNCTION InitDateCache(theCache: DateCachePtr): OSErr;
```

`theCache`	A pointer to a record of type `DateCacheRecord`. This parameter can be a local variable, a pointer, or a locked handle.

DESCRIPTION

You must call `InitDateCache` to initialize the date cache record before using either the `StringToDate` or `StringToTime` functions. You must pass a pointer to a date cache record. You have to declare the record as a variable or allocate it in the heap.

If you are writing an application that allows the use of global variables, you can make your date cache record a global variable and initialize it once, when you perform other global initialization.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `InitDateCache` function are

Trap macro	Selector
`_ScriptUtil`	$8204 FFF8

RESULT CODES

`noErr`	0	No error
`fatalDateTime`	–32768	A miscellaneous fatal error occurred, usually a failure in a call to get a resource

SEE ALSO

`InitDateCache` calls the `GetResource` and `LoadResource` routines and it can also return the error codes they produce. These routines and their return values are described in the book *Inside Macintosh: Operating System Utilities*.

StringToDate

The `StringToDate` function parses a string for a date and converts the date information into values in a date-time record. It expects a date specification, in a format defined by the current script, at the beginning of the string. It returns a status value that indicates the confidence level for the success of the conversion.

```
FUNCTION StringToDate(textPtr: Ptr; textLen: LongInt;
                  theCache: DateCachePtr; VAR lengthUsed: LongInt;
                  VAR dateTime: LongDateRec): StringToDateStatus;
```

`textPtr`	A pointer to the text string to be parsed.
`textLen`	The number of bytes in the text string.
`theCache`	A pointer to the date cache record initialized by the `InitDateCache` function with data that is used during the conversion process.
`lengthUsed`	On output, contains the number of bytes of the string that were parsed for the date.
`dateTime`	On output, this `LongDateRec` record contains the year, month, day, and day of the week parsed for the date.

DESCRIPTION

`StringToDate` parses the text string until it has finished finding all date information or until it has examined the number of bytes specified by `textLen`. It returns a status value that indicates the confidence level for the success of the conversion. For an overview of how this function operates, see the section "Converting Formatted Date and Time Strings Into Internal Numeric Representations" beginning on page 5-31.

Note that `StringToDate` fills in only the year, month, day, and day of the week; `StringToTime` fills in the hour, minute, and second. You can use these two routines sequentially to fill in all of the values in a `LongDateRec` record.

`StringToDate` assigns to its `lengthUsed` parameter the number of bytes that it uses to parse the date; use this value to compute the starting location of the text that you can pass to `StringToTime` (or you can use them in reverse order).

When one of the date components is missing, such as the year, the current date value is used as a default. If the value of the input year is less than 100, `StringToDate` determines the year as described on page 5-32.

`StringToDate` returns a value of type `StringToDateStatus`, which is a set of bit values that indicate confidence levels, with higher numbers indicating low confidence in how closely the input string matched what the routine expected. For example, specifying a date with nonstandard separators may work, but it returns a message indicating that the separator was not standard. The possible values of this type are described in Table 5-5 on page 5-33.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StringToDate` function are

Trap macro	Selector
`_ScriptUtil`	$8214 FFF6

StringToTime

The `StringToTime` function parses a string for a time specification and converts the date information into values in a date-time record. At the beginning of the string, it expects a time specification in a format defined by the current script. It returns a status value that indicates the confidence level for the success of the conversion.

```
FUNCTION StringToTime(textPtr: Ptr; textLen: LongInt;
                  theCache: DateCachePtr; VAR lengthUsed: LongInt;
                  VAR dateTime: LongDateRec): StringToDateStatus;
```

`textPtr`	A pointer to the text string to be parsed.
`textLen`	The number of bytes in the text string.
`theCache`	A pointer to the date cache record initialized by the `InitDateCache` function with data that is used during the conversion process.
`lengthUsed`	On output, contains the number of bytes of the string that were parsed for the time.
`dateTime`	On output, this `LongDateRec` record contains the hour, minute, and second values that were parsed for the time.

DESCRIPTION

`StringToTime` parses the string until it has finished finding all time information or until it has examined the number of bytes specified by `textLen`. It returns a status value that indicates the confidence level for the success of the conversion.

Note that `StringToTime` fills in only the hour, minute, and second; `StringToDate` fills in the year, month, day, and day of the week. You can use these two routines sequentially to fill in all of the values in a `LongDateRec` record.

`StringToTime` assigns to its `lengthUsed` parameter the number of bytes that it used to parse the date.

StringToTime returns the same status value indicator type as does StringToDate: a set of bit values that indicate confidence levels, with higher numbers indicating low confidence in how closely the input string matched what the routine expected. The possible values of this type are described in Table 5-5 on page 5-33.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the StringToTime function are

Trap macro	Selector
_ScriptUtil	$8214 FFF4

Converting Numeric Representations Into Date and Time Strings

This section describes the routines that you can use to convert numeric representations of date and time values into formatted strings. The numeric representation used in these routines is the standard date-time representation: a 32-bit integer value that is returned by the GetDateTime routine. This is a long integer value that represents the number of seconds between midnight, January 1, 1904, and the time at which GetDateTime was called, as described in the book *Inside Macintosh: Operating System Utilities*.

- The DateString procedure converts a date in the standard date-time representation into a string, making use of the date formatting information from a specified resource. If you specify NIL as the value of the resource handle parameter, DateString uses information from the current script.
- The TimeString procedure converts a time in the standard date-time representation into a string, making use of the time formatting information from a specified resource. If you specify NIL as the value of the resource handle parameter, TimeString uses information from the current script.

DateString

The DateString procedure converts a date in the standard date-time representation into a Pascal string, making use of the date formatting information in the specified resource.

```
PROCEDURE DateString (dateTime: LongInt; longFlag: Boolean;
                      VAR result: Str255; intlParam: Handle );
```

dateTime	The date-time value in the representation returned by the GetDateTime procedure.

`longFlag`	A flag that indicates the desired format for the date string. This is one of the three values defined as the `DateForm` type.
`result`	On output, contains the string representation of the date in the format indicated by the `longFlag` parameter.
`intlParam`	A handle to a numeric-format or a long-date-format resource that specifies date formatting information for use in the conversion. The numeric-format (`'itl0'`) resource specifies the short date formats and the long-date-format (`'itl1'`) resource specifies the long date formats.

DESCRIPTION

`DateString` converts the long integer representation of date and time in the `dateTime` parameter into a Pascal string representation of the date. You can call the `GetDateTime` function to get the date-time value. `GetDateTime` is described in the book *Inside Macintosh: Operating System Utilities*.

The string produced by `DateString` is in one of three standard date formats used on the Macintosh, depending on which of the three `DateForm` values that you specify for the `longFlag` parameter: `shortDate`, `abbrevDate`, or `longDate`. The information in the supplied resource defines how month and day names are written and provides for calendars with more than 7 days and more than 12 months. For the Roman script system's resource, the date January 31, 1991, produces the following three strings:

Dateform value	Date string produced
`shortDate`	1/31/92
`abbrevDate`	Fri, Jan 31, 1992
`longDate`	Friday, January 31, 1992

`DateString` formats its data according to the information in the specified numeric-format resource (for short date formats) or long-date-format resource (for long date formats). If you specify `shortDate`, the `intlParam` value should be the handle to a numeric-format resource; if you specify `abbrevDate` or `longDate`, it should be the handle to a long-date-format resource. If the `intlParam` value is `NIL`, `DateString` uses the appropriate resource from the current script.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `DateString` procedure are

Trap macro	Selector
`_Pack6`	$000E

TimeString

The `TimeString` procedure converts a time in the standard date-time representation into a string, making use of the time formatting information in the specified resource.

```
PROCEDURE TimeString (dateTime: LongInt; wantSeconds: Boolean;
                      VAR result: Str255; intlParam: Handle);
```

`dateTime`	The date-time value in the representation returned by the Operating System procedure `GetDateTime`.
`wantSeconds`	A flag that indicates whether the seconds are to be included in the resulting string.
`result`	On output, contains the string representation of the time.
`intlParam`	A handle to a numeric-format (`'itl0'`) resource that specifies time formatting information for use in the conversion.

DESCRIPTION

`TimeString` converts the long integer representation of date and time in the `dateTime` parameter into a Pascal string representation of the time. You can call the `GetDateTime` function to get the date-time value. `GetDateTime` is described in the book *Inside Macintosh: Operating System Utilities*.

`TimeString` produces a string that includes the seconds if you set the `wantSeconds` parameter to `TRUE`.

`TimeString` formats its data according to the information in the numeric-format resource specified in the `intlParam` parameter. If this value is `NIL`, `TimeString` uses the numeric-format resource from the current script. The numeric-format resource specifies whether or not to use leading zeros for the time values, whether to use a 12- or 24-hour time cycle, and how to specify morning or evening if a 12-hour time cycle is used.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `TimeString` procedure are

Trap macro	Selector
`_Pack6`	$0010

Converting Long Date and Time Values Into Strings

This section describes two procedures that use the `LongDateTime` data type in their conversions. This is a 64-bit, signed representation of the number of seconds since Jan. 1, 1904, which allows coverage of a much longer span of time (plus or minus approximately 30,000 years) than the standard, 32-bit representation. `LongDateTime` values are described in the book *Inside Macintosh: Operating System Utilities*.

- The `LongDateString procedure` converts a date in `LongDateTime` representation into a string, making use of the date formatting information from a specified resource. If you specify `NIL` as the value of the resource handle parameter, `LongDateString` uses information from the current script.
- The `LongTimeString procedure` converts a time in `LongDateTime` representation into a string, making use of the date formatting information from a specified resource. If you specify `NIL` as the value of the resource handle parameter, `LongTimeString` uses information from the current script.

LongDateString

The `LongDateString` procedure converts a date that is specified as a `LongDateTime` value into a Pascal string, making use of the date formatting information in the specified resource.

```
PROCEDURE LongDateString(VAR dateTime: LongDateTime;
                         longFlag: DateForm;
                         VAR result: Str255; intlParam: Handle);
```

`dateTime`	A 64-bit, signed representation of the number of seconds since Jan. 1, 1904.
`longFlag`	A flag that indicates the desired format for the date string. This is one of the three values defined as the `DateForm` type.
`result`	On output, contains the string representation of the date in the format indicated by the `longFlag` parameter.
`intlParam`	A handle to a numeric-format or long-date-format resource that specifies date formatting information for use in the conversion. The numeric-format (`'itl0'`) resource specifies the short date formats and the long-date-format (`'itl1'`) resource specifies the long date formats.

DESCRIPTION

`LongDateString` converts the `LongDateTime` value in the `dateTime` parameter into a Pascal string representation of the date. You can use the `LongSecondsToDate` and `LongDateToSeconds` procedures, which are described in the book *Inside Macintosh: Operating System Utilities*, to convert between the `LongDateRec` (as produced by the `StringToDate` function) and `LongDateTime` data types.

The string produced by `LongDateString` is in one of three standard date formats used on the Macintosh, depending on which of the three `DateForm` values that you specify for the `longFlag` parameter: `shortDate`, `abbrevDate`, or `longDate`. The information in the supplied resource defines how month and day names are written and provides for calendars with more than 7 days and more than 12 months. For the U.S. resource, the date January 31, 1991, produces the following three strings:

`DateForm` value	**Date string produced**
`shortDate`	1/31/92
`abbrevDate`	Fri, Jan 31, 1992
`longDate`	Friday, January 31, 1992

`LongDateString` formats its data according to the information in the specified numeric-format resource (for short date formats) or long-date-format resource (for long date formats). If you specify `shortDate`, the `intlParam` value should be the handle to a numeric-format resource; if you specify `abbrevDate` or `longDate`, it should be the handle to a long-date-format resource. If the `intlParam` value is `NIL`, `LongDateString` uses the resource from the current script.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LongDateString` procedure are

Trap macro	**Selector**
`_Pack6`	$0014

LongTimeString

The `LongTimeString` procedure converts a time that is specified as a `LongDateTime` value into a Pascal string, making use of the time formatting information in the specified resource.

```
PROCEDURE LongTimeString(VAR dateTime: LongDateTime;
                         wantSeconds: Boolean;
                         VAR result: Str255; intlParam: Handle);
```

`dateTime`	A 64-bit, signed representation of the number of seconds since Jan. 1, 1904.
`wantSeconds`	A flag that indicates whether the seconds are to be included in the resulting string.

`result`	On output, contains the string representation of the time.
`intlParam`	A handle to a numeric-format (`'itl0'`) resource that specifies time formatting information for use in the conversion.

DESCRIPTION

`LongTimeString` converts the `LongDateTime` value in the `dateTime` parameter into a Pascal string representation of the time. You can use the `LongSecondsToDate` and `LongDateToSeconds` procedures, which are described in the book *Inside Macintosh: Operating System Utilities,* to convert between the `LongDateRec` (as produced by the `StringToTime` function) and `LongDateTime` data types.

`LongTimeString` produces a string that includes the seconds if you set the `wantSeconds` parameter to `TRUE`.

`LongTimeString` formats its data according to the information in the numeric-format resource specified in the `intlParam` parameter. If this value is `NIL`, `LongTimeString` uses the numeric-format resource from the current script. The numeric-format resource specifies whether or not to use leading zeros for the time values, whether to use a 12- or 24-hour time cycle, and how to specify morning or evening if a 12-hour time cycle is used.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `LongTimeString` procedure are

Trap macro	Selector
`_Pack6`	$0016

Converting Between Integers and Strings

This section describes routines that allow you to convert between string and numeric representations of numbers. Unless patched by a script system with different rules, these two routines assume that you are using standard numeric token processing, meaning that the Roman script system number processing rules are used.

- The `NumToString` procedure converts a long integer value to a string representation of it as a base-10 number.
- The `StringToNum` procedure converts a string representation of a base-10 number into a long integer value.

For routines that make use of the token-processing information that is found in the tokens (`'itl4'`) resource of script systems for converting numbers, see the section "Using Number Format Specification Strings for International Number Formatting," which begins on page 5-94, and the section "Converting Between Strings and Floating-Point Numbers," which begins on page 5-98.

NumToString

The `NumToString` procedure converts a long integer value into a Pascal string.

```
PROCEDURE NumToString (theNum: LongInt; VAR theString: Str255);
```

`theNum`	A long integer value.
`theString`	On output, contains the Pascal string representation of the number.

DESCRIPTION

`NumToString` creates a string representation of `theNum` as a base-10 value and returns the result in `theString`.

If the value of the number in the parameter `theNum` is negative, the string begins with a minus sign; otherwise, the sign is omitted. Leading zeros are suppressed, except that a value of 0 produces the string "0". `NumToString` does not include thousand separators or decimal points in its formatted output.

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `NumToString` procedure are

Trap macro	Selector
`_Pack7`	$0000

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to the length byte that precedes `theString`
D0	the long integer value to be converted

Registers on exit

D0	pointer to the length byte that precedes `theString`

StringToNum

The `StringToNum` procedure converts the Pascal string representation of a base-10 number into a long integer value.

```
PROCEDURE StringToNum (theString: Str255; VAR theNum: LongInt);
```

`theString`	A Pascal string representation of a base-10 number.
`theNum`	On output, contains the numeric value.

DESCRIPTION

`StringToNum` converts the base-10 numeric string in the `theString` parameter to the corresponding long integer value and returns the result in the parameter `theNum`. The numeric string can be padded with leading zeros or with a sign.

The 32-bit result is negated if the string begins with a minus sign. Integer overflow occurs if the magnitude is greater than or equal to 2 raised to the 31st power. `StringToNum` performs the negation using the two's complement method: the state of each bit is reversed and then 1 is added to the result. For example, here are possible results produced by `StringToNum`:

Value of `theString`	Value returned in `theNum`
"–23"	–23
"–0"	0
"055"	55
"2147483648" (magnitude is 2^31)	–2147483648
"–2147483648"	–2147483648
"4294967295" (magnitude is 2^32–1)	–1
"–4294967295"	1

`StringToNum` does not check whether the characters in the string are between 0 and 9; instead, it takes advantage of the fact that the ASCII values for these characters are $30 through $39, and masks the last four bits for use as a digit. For example, `StringToNum` converts 2: to the number 30 since the character code for the colon (:) is $3A. Because `StringToNum` operates this way, spaces are treated as zeros (the character code for a space is $20), and other characters do get converted into numbers. For example, the character codes for 'C', 'A', and 'T' are $43, $41, and $54 respectively, producing these results:

Value of `theString`	Value returned in `theNum`
'CAT'	314
'+CAT'	314
'–CAT'	–314

Note

One consequence of this conversion method is that `StringToNum` does not ignore thousand separators (the "," character in the United States), which can lead to improper conversions. It is a good idea to ensure that all characters in `theString` are valid digits before you call `StringToNum`. ◆

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StringToNum` procedure are

Trap macro	Routine selector
`_Pack7`	$0001

The registers on entry and exit for this routine are

Registers on entry

A0	pointer to the length byte that precedes `theString`

Registers on exit

D0	the long word value

Using Number Format Specification Strings for International Number Formatting

To allow for all of the international variations in numeric presentation styles, you need to include in your routine calls a number parts table from a tokens (`'itl4'`) resource. You can usually use the number parts table in the standard tokens resource that is supplied with the system. You also need to define the format of input and output numeric strings, including which characters (if any) to use as thousand separators, whether to indicate negative values with a minus sign or by enclosing the number in parentheses, and how to display zero values. These details are specified in number format specification strings, the syntax of which is described in the section "Using Number Format Specification Strings," beginning on page 5-39.

To make it possible to map a number that was formatted for one specification into another format, the Macintosh Operating System defines an internal numeric representation that is independent of region, language, and other multicultural considerations: the `NumFormatStringRec` record. This record is created from a number format specification string that defines the appearance of numeric strings. Its use is summarized in Figure 5-12 on page 5-37.

In brief, what you have to do is create a number format specification string that you want to use and convert that string into a `NumFormatStringRec` record. The Text Utilities include two routines for this purpose:

- The `StringToFormatRec` function converts a number format specification string into a `NumFormatStringRec` record.
- The `FormatRecToString` function converts the internal representation in a `NumFormatStringRec` record into a number format specification string, which can be viewed and modified.

StringToFormatRec

The `StringToFormatRec` function creates a number format specification string record from a number format specification string that you supply in a Pascal string.

```
FUNCTION StringToFormatRec(inString: Str255;
               partsTable: NumberParts;
               VAR outString: NumFormatStringRec): FormatStatus;
```

`inString` A Pascal string that contains the number formatting specification.

`partsTable`
A record usually obtained from the tokens (`'itl4'`) resource that shows the correspondence between generic number part separators (tokens) and their localized version (for example, a thousand separator is a comma in the United States and a decimal point in France).

`outString` On output, this `NumFormatStringRec` record contains the values that form the internal representation of the format specification. The format of the data in this record is private.

DESCRIPTION

`StringToFormatRec` converts a number format specification string into the internal representation contained in a number format string record. It uses information in the current script's tokens resource to determine the components of the number. `StringToFormatRec` checks the validity both of the input format string and of the number parts table (since this table can be programmed by the application). `StringToFormatRec` ignores spurious characters.

The `inString` parameter contains a number format specification string that specifies how numbers appear. This string contains up to three specifications, separated by semicolons. The positive number format is specified first, the negative number format is second, and the zero number format is last. If the string contains only one part, that is the format of all three types of numbers. If the string contains two parts, the first part is the format for positive and zero number values, and the second part is the format for negative numbers. The syntax for the number format specification strings is described in detail in "Using Number Format Specification Strings," which begins on page 5-39.

`StringToFormatRec` returns a value of type `FormatStatus` that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`, the values of which are described in Table 5-6 on page 5-38.

IMPORTANT

Be sure to cast the result of `StringToFormatRec` to a type `FormatResultType` before working with it. ▲

SPECIAL CONSIDERATIONS

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

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StringToFormatRec` function are

Trap macro	Selector
`_ScriptUtil`	$820C FFEC

SEE ALSO

For comprehensive details on the number parts table, see the appendix "International Resources" in this book.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` procedure, which is described in the chapter "Script Manager" in this book.

FormatRecToString

The `FormatRecToString` function converts an internal representation of number formatting information into a number format specification string, which can be displayed and modified.

```
FUNCTION FormatRecToString(myFormatRec: NumFormatStringRec;
                  partsTable: NumberParts; VAR outString: Str255;
                  VAR positions: TripleInt): FormatStatus;
```

`myFormatRec`
: The internal representation of number formatting information, as created by a previous call to the `StringToFormatRec` function.

`partsTable`	A record obtained from the tokens (`'itl4'`) resource that shows the correspondence between generic number part separators (tokens) and their localized version (for example, a thousand separator is a comma in the United States and a decimal point in France).
`outString`	On output, contains the number format specification string.
`positions`	An array that specifies the starting position and length of each of the three possible format strings (positive, negative, or zero) in the number format specification string. Semicolons are used as separators in the string.

DESCRIPTION

`FormatRecToString` is the inverse operation of `StringToFormatRec`. The internal representation of the formatting information in `myFormatRec` must have been created by a prior call to the `StringToFormatRec` function. The information in the number parts table specifies how to build the string representation.

The output number format specification string in `outString` specifies how numbers appear. This string contains three parts, which are separated by semicolons. The first part is the positive number format, the second is the negative number format, and the third part is the zero number format. The syntax for this string is described in detail in "Using Number Format Specification Strings," which begins on page 5-39.

The `positions` parameter is an array of three integers (a `TripleInt` value), which specifies the starting position in `outString` of each of three formatting specifications:

Array entry	**What its value specifies**
`positions[fPositive]`	the index in `outString` of the first byte of the formatting specification for positive number values
`positions[fNegative]`	the index in `outString` of the first byte of the formatting specification for negative number values
`positions[fZero]`	the index in `outString` of the first byte of the formatting specification for zero number values

`FormatRecToString` returns a value of type `FormatStatus` that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`, the values of which are described in Table 5-6 on page 5-38.

IMPORTANT
Be sure to cast the result of `FormatRecToString` to a type `FormatResultType` before working with it. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `FormatRecToString` function are

Trap macro	**Selector**
`_ScriptUtil`	$8210 FFEA

SEE ALSO

For comprehensive details on the number parts table, see the appendix "International Resources" in this book.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` procedure, which is described in the chapter "Script Manager" in this book.

Converting Between Strings and Floating-Point Numbers

Once you have created a `NumFormatStringRec` record that specifies how numbers are represented, as described in "Using Number Format Specification Strings for International Number Formatting," which begins on page 5-94, you can use two other Text Utilities routines to convert between string and floating-point representations of numbers. Floating-point numbers are stored in standard Apple (SANE) format.

- The `StringToExtended` function converts the string representation of a number into a floating-point number, using a `NumFormatStringRec` record to specify how the input number string is formatted.
- The `ExtendedToString` function converts a floating-point number into a string that can be presented to the user, using a `NumFormatStringRec` record to specify how the output number string is formatted.

StringToExtended

The `StringToExtended` function converts a string representation of a number into a floating-point number.

```
FUNCTION StringToExtended(source: Str255;
                          myFormatRec: NumFormatStringRec;
                          partsTable: NumberParts;
                          VAR x: Extended80): FormatStatus;
```

`source`	A Pascal string that contains the string representation of a number.
`myFormatRec`	The internal representation of the formatting information for numbers, as produced by the `StringToFormatRec` function.
`partsTable`	A record obtained from the tokens (`'itl4'`) resource that shows the correspondence between generic number part separators (tokens) and their localized version (for example, a thousand separator is a comma in the United States and a decimal point in France).
`x`	On output, contains the 80-bit SANE representation of the floating-point number.

DESCRIPTION

`StringToExtended` uses the internal representation of number formatting information that was created by a prior call to `StringToFormatRec` to parse the input number string. It uses the number parts table to determine the components of the number string that is being converted. `StringToExtended` parses the string and then converts the string to a simple form, stripping nondigits and replacing the decimal point before converting it into a floating-point number. If the input string does not match any of the patterns, then `StringToExtended` parses the string as well as it can and returns a confidence level result that indicates the parsing difficulties.

`StringToFormatRec` returns a value of type `FormatStatus` that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`, the values of which are described in Table 5-6 on page 5-38.

IMPORTANT

Be sure to cast the result of `StringToExtended` to a type `FormatResultType` before working with it. ▲

SPECIAL CONSIDERATIONS

`StringToExtended` returns an 80-bit, not a 96-bit, representation.

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `StringToExtended` function are

Trap macro	Selector
`_ScriptUtil`	$8210 FFE6

SEE ALSO

For comprehensive details on the number parts table, see the description of the tokens (`'itl4'`) resource in the appendix “International Resources” in this book.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` procedure, which is described in the chapter “Script Manager” in this book.

ExtendedToString

The `ExtendedToString` function converts an internal floating-point representation of a number into a string that can be presented to the user.

```
FUNCTION ExtendedToString(x: Extended80;
                          myFormatRec: NumFormatStringRec;
                          partsTable: NumberParts;
                          VAR outString: Str255): FormatStatus;
```

`x`	A floating-point value in 80-bit SANE representation.
`myFormatRec`	The internal representation of the formatting information for numbers, as produced by the `StringToFormatRec` function.
`partsTable`	A record obtained from the tokens (`'itl4'`) resource that shows the correspondence between generic number part separators (tokens) and their localized version (for example, a thousand separator is a comma in the United States and a decimal point in France).
`outString`	On output, contains the number formatted according to the information in `myFormatRec`.

DESCRIPTION

`ExtendedToString` creates a string representation of a floating-point number, using the formatting information in the `myFormatRec` parameter (which was created by a previous call to `StringToFormatRec`) to determine how the number should be formatted for output. It uses the number parts table to determine the component parts of the number string.

`StringToFormatRec` returns a value of type `FormatStatus` that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`, the values of which are described in Table 5-6 on page 5-38.

IMPORTANT

Be sure to cast the result of `ExtendedToString` to a type `FormatResultType` before working with it. ▲

ASSEMBLY-LANGUAGE INFORMATION

The trap macro and routine selector for the `ExtendedToString` function are

Trap macro	Selector
`_ScriptUtil`	$8210 FFE8

SEE ALSO

For comprehensive details on the number parts table, see the description of the tokens (`'itl4'`) resource in the appendix "International Resources" in this book.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` procedure, which is described in the chapter "Script Manager" in this book.

Summary of Text Utilities

Pascal Summary

Constants

```
CONST
         {StringToDate and StringToTime status values }
   longDateFound     = 1;     {mask to long date found}
   leftOverChars     = 2;     {mask to warn of left over chars}
   sepNotIntlSep     = 4;     {mask to warn of non-standard separators}
   fieldOrderNotIntl = 8;     {mask to warn of non-standard field order}
   extraneousStrings = 16;    {mask to warn of unparsable strings in text}
   tooManySeps       = 32;    {mask to warn of too many separators}
   sepNotConsistent  = 64;    {mask to warn of inconsistent separators}
   fatalDateTime     = $8000; {mask to a fatal error}
   tokenErr = $8100;          {mask for 'tokenizer err encountered'}
   cantReadUtilities = $8200; {mask for can’t access needed resource}
   dateTimeNotFound = $8400;  {mask for date or time not found}
   dateTimeInvalid = $8800;   {mask for date/time format not valid}

         {Constants for truncWhere argument in TruncString and TruncText}
   truncEnd       = 0;        {truncate at end}
   truncMiddle    = $4000;    {truncate in middle}

         {Constants for TruncString and TruncText results}
   NotTruncated   = 0;        {no truncation was necessary}
   Truncated      = 1;        {truncation performed}
   TruncErr       = -1;       {general error}

         {Special language code values for Language Order}
systemCurLang  = -2; { current language for system script (from 'itlb')}
systemDefLang  = -3; { default language for system script (from 'itlm')}
currentCurLang = -4; { current language for current script (from 'itlb')}
currentDefLang = -5; { default language for current script (from 'itlm')}
scriptCurLang  = -6; { current lang for specified script (from 'itlb')}
scriptDefLang  = -7; { default language for specified script (from 'itlm')}
```

Data Types

```
TYPE

FormatStatus = Integer;

TruncCode = Integer;

DateForm = (shortDate,longDate,abbrevDate);

FormatResultType =
(fFormatOK,fBestGuess,fOutOfSynch,fSpuriousChars,fMissingDelimiter,
   fExtraDecimal,fMissingLiteral,fExtraExp,fFormatOverflow,fFormStrIsNAN,
   fBadPartsTable,fExtraPercent,fExtraSeparator,fEmptyFormatString);

FormatClass = (fPositive,fNegative,fZero);

StyledLineBreakCode = {BreakWord, BreakChar, BreakOverflow};

DateCacheRecord =
PACKED RECORD
   hidden: ARRAY [0..255] OF Integer;{only for temporary use}
END;

DateCachePtr = ^DateCacheRecord;

NumFormatStringRec =
PACKED RECORD
   fLength: Byte;
   fVersion: Byte;
   data: PACKED ARRAY [0..253] OF SignedByte;   {private data}
END;

FVector =
RECORD
   start: Integer;
   length: Integer
END;

TripleInt = ARRAY[0..2] OF FVector;          {index by [fPositive..fZero]}
```

```
OffPair =
RECORD
   offFirst: Integer;
   offSecond: Integer;
END;

OffsetTable = ARRAY[0..2] OF OffPair;

ScriptRunStatus =
RECORD
   script: SignedByte;
   variant: SignedByte;
END;
```

Routines

Defining and Specifying Strings

```
FUNCTION NewString            (theString: Str255): StringHandle;
PROCEDURE SetString           (theString: StringHandle; strNew: Str255);
FUNCTION GetString            (stringID: Integer): StringHandle;
PROCEDURE GetIndString        (VAR theString: Str255; strListID: Integer;
                               index: Integer);
```

Comparing Strings for Equality

```
FUNCTION EqualString          (aStr, bStr: Str255;
                               caseSens, diacSens: Boolean): Boolean;
FUNCTION IdenticalString      (aStr, bStr: Str255;
                               itl2Handle: Handle): Integer;
FUNCTION IdenticalText        (aPtr, bPtr: Ptr; aLen, bLen: Integer;
                               itl2Handle: Handle): Integer;
```

Determining Sorting Order for Strings in Different Languages

```
FUNCTION ScriptOrder          (script1, script2: ScriptCode): Integer;
FUNCTION LanguageOrder        (lang1, lang2: LangCode): Integer;
FUNCTION StringOrder          (aStr, bStr: Str255; aScript, bScript:
                               ScriptCode; aLang, bLang: LangCode): Integer;
FUNCTION TextOrder            (aPtr, bPtr: Ptr; aLen, bLen: Integer;
                               aScript, bScript: ScriptCode;
                               aLang, bLang: LangCode): Integer;
```

Determining Sorting Order for Strings in the Same Language

```
FUNCTION RelString          (aStr, bStr: Str255;
                             caseSens, diacSens: Boolean): Integer;
FUNCTION CompareString      (aStr, bStr: Str255;
                             itl2Handle: Handle): Integer;
FUNCTION CompareText        (aPtr, bPtr: Ptr; aLen, bLen: Integer): Integer;
```

Modifying Characters and Diacritical Marks

```
PROCEDURE UpperString       (VAR theString: Str255; diacSens: Boolean);
PROCEDURE LowercaseText     (textPtr: Ptr; len: Integer;
                             script: ScriptCode);
PROCEDURE UppercaseText     (textPtr: Ptr; len: Integer;
                             script: ScriptCode);
PROCEDURE StripDiacritics   (textPtr: Ptr; len: Integer;
                             script: ScriptCode);
PROCEDURE UppercaseStripDiacritics
                            (textPtr: Ptr; len: Integer;
                             script: ScriptCode);
```

Truncating Strings

```
FUNCTION TruncString        (width: Integer; VAR theString: Str255;
                             truncWhere: TruncCode): Integer;
FUNCTION TruncText          (width: Integer; textPtr: Ptr;
                             VAR length: Integer;
                             truncWhere: TruncCode): Integer;
```

Searching for and Replacing Strings

```
FUNCTION ReplaceText        (baseText, substitutionText: Handle;
                             key: Str15): Integer;
FUNCTION Munger             (h: Handle; offset: LongInt;
                             ptr1: Ptr; len1: LongInt;
                             ptr2: Ptr; len2: LongInt): LongInt;
```

Working With Word, Subscript, and Line Boundaries

```
PROCEDURE FindWordBreaks    (textPtr: Ptr; textLength: Integer;
                             offset: Integer; leadingEdge: Boolean;
                             nBreaks: NBreakTablePtr;
                             VAR offsets:OffsetTable );
```

```
FUNCTION StyledLineBreak    (textPtr: Ptr; textLen: LongInt;
                             textStart, textEnd, flags: LongInt;
                             VAR textWidth: Fixed;
                             VAR textOffset: LongInt): StyledLineBreakCode;
FUNCTION FindScriptRun      (textPtr: Ptr; textLen: LongInt;
                             VAR lenUsed: LongInt): ScriptRunStatus;
```

Converting Date and Time Strings Into Numeric Representations

```
FUNCTION InitDateCache      (theCache: DateCachePtr): OSErr;
FUNCTION StringToDate       (textPtr: Ptr; textLen: LongInt;
                             theCache: DateCachePtr;
                             VAR lengthUsed: LongInt;
                             VAR dateTime: LongDateRec): StringToDateStatus;
FUNCTION StringToTime       (textPtr: Ptr; textLen: LongInt;
                             theCache: DateCachePtr;
                             VAR lengthUsed: LongInt;
                             VAR dateTime: LongDateRec): StringToDateStatus;
```

Converting Numeric Representations Into Date and Time Strings

```
PROCEDURE DateString        (dateTime: LongInt; longFlag: DateForm;
                             VAR result: Str255; intlHandle: Handle);
PROCEDURE TimeString        (dateTime: LongInt; wantSeconds: Boolean;
                             VAR result: Str255; intlHandle: Handle);
```

Converting Long Date and Time Values Into Strings

```
PROCEDURE LongDateString    (VAR dateTime: LongDateTime; longFlag: DateForm;
                             VAR result: Str255; intlHandle: Handle);
PROCEDURE LongTimeString    (VAR dateTime: LongDateTime;
                             wantSeconds:Boolean; VAR result: Str255;
                             intlHandle: Handle);
```

Converting Between Integers and Strings

```
PROCEDURE NumToString       (theNum: LongInt; VAR theString: Str255);
PROCEDURE StringToNum       (theString: Str255; VAR theNum: LongInt);
```

Using Number Format Specification Strings for International Number Formatting

```
FUNCTION StringToFormatRec  (inString: Str255; partsTable: NumberParts;
                             VAR outString: NumFormatString): FormatStatus;
```

```
FUNCTION FormatRecToString  (myFormatRec: NumFormatString;
                             partsTable: NumberParts;
                             VAR outString: Str255;
                             VAR positions: TripleInt): FormatStatus;
```

Converting Between Strings and Floating-Point Numbers

```
FUNCTION StringToExtended   (source: Str255; myFormatRec: NumFormatString;
                             partsTable: NumberParts;
                             VAR x: Extended80): FormatStatus;
FUNCTION ExtendedToString   (x: Extended80; myFormatRec: NumFormatString;
                             partsTable: NumberParts;
                             VAR outString: Str255): FormatStatus;
```

C Summary

Constants

```
enum {      /*StringToDate and StringToTime status values*/
   longDateFound = 1;         /*mask to long date found*/
   leftOverChars = 2;         /*mask to warn of left over chars*/
   sepNotIntlSep = 4;         /*mask to warn of non-standard separators*/
   fieldOrderNotIntl = 8;     /*mask to warn of non-standard field order*/
   extraneousStrings = 16;    /*mask to warn of unparsable strings */
   tooManySeps = 32;          /*mask to warn of too many separators*/
   sepNotConsistent = 64;     /*mask to warn of inconsistent separators*/
   fatalDateTime = 0x8000;    /*mask to a fatal error*/
   tokenErr = 0x8100;         /*mask for 'tokenizer err encountered'*/
   cantReadUtilities = 0x8200;/*mask for can’t access needed resource*/
   dateTimeNotFound = 0x8400; /*mask for date or time not found*/
   dateTimeInvalid = 0x8800;  /*mask for date/time format not valid*/
};

enum {   /*constants for truncWhere argument in TruncString and TruncText*/
   truncEnd = 0,              /*truncate at end*/
   truncMiddle = 0x4000,      /*truncate in middle*/
};
```

```
enum {   /*constants for TruncString and TruncText results*/
   notTruncated = 0,       /*no truncation was necessary*/
   truncated = 1,          /*truncation performed*/
   truncErr = -1,          /*general error*/
};

enum {   /*special language code values for Language Order*/
   systemCurLang = -2,  /*current lang for system script (from 'itlb')*/
   systemDefLang = -3,  /*default lang for system script (from 'itlm')*/
   currentCurLang = -4, /*current lang for current script (from 'itlb')*/
   currentDefLang = -5, /*default lang for current script (from 'itlm')*/
   scriptCurLang = -6,  /*current lang for specified script (from 'itlb')*/
   scriptDefLang = -7,  /*default lang for specified script (from 'itlm')*/
};

enum {
   BreakWord,
   BreakChar,
   BreakOverflow
};

enum {
   fPositive,
   fNegative,
   fZero
};

enum{
   fFormatOK,
   fBestGuess,
   fOutOfSynch,
   fSpuriousChars,
   fMissingDelimiter,
   fExtraDecimal,
   fMissingLiteral,
   fExtraExp,
   fFormatOverflow,
   fFormStrIsNAN,
   fBadPartsTable,
   fExtraPercent,
   fExtraSeparator,
   fEmptyFormatString
};
```

```
enum {
   shortDate,
   longDate,
   abbrevDate
};
```

Types

```
typedef short StringToDateStatus;

typedef unsigned char StyledLineBreakCode;

typedef unsigned char FormatClass;

typedef short TruncCode;

typedef unsigned char FormatResultType;

typedef unsigned char DateForm;

struct DateCacheRecord {
   short hidden[256];            /*only for temporary use*/
};

typedef struct DateCacheRecord DateCacheRecord;
typedef DateCacheRecord *DateCachePtr;

struct NumFormatString {
   char fLength;
   char fVersion;
   char data[254];               /*private data*/
};

typedef struct NumFormatString NumFormatStringRec;
struct FVector {
   short start;
   short length;
};

typedef struct FVector FVector;
typedef FVector TripleInt[3];    /* index by [fPositive..fZero] */

struct ScriptRunStatus {
   char script;
   char variant;
};

typedef struct ScriptRunStatus ScriptRunStatus;

struct OffPair {
   short offFirst;
   short offSecond;
};

typedef struct OffPair OffPair;
typedef OffPair OffsetTable[3];
```

Routines

Defining and Specifying Strings

```
pascal StringHandle NewString
                            (ConstStr255Param theString);
pascal void SetString       (StringHandle theString,
                             ConstStr255Param strNew);
pascal StringHandle GetString
                            (short stringID);
pascal void GetIndString    (Str255 theString, short strListID,
                             short index);
```

Comparing Strings for Equality

```
pascal Boolean EqualString  (ConstStr255Param aStr, ConstStr255Param bStr,
                             Boolean caseSens, Boolean diacSens );
pascal short IdenticalString
                            (ConstStr255Param aStr, ConstStr255Param bStr,
                             Handle itl2Handle);
pascal short IdenticallText (const void *aPtr, const void *bPtr,
                             short aLen, short bLen, Handle itl2Handle);
```

Determining Sorting Order for Strings in Different Languages

```
pascal short ScriptOrder    (ScriptCode script1, ScriptCode script2);
pascal short LanguageOrder  (LangCode language1, LangCode language2);
pascal short StringOrder    (ConstStr255Param aStr, ConstStr255Param bStr,
                             ScriptCode aScript, ScriptCode bScript,
                             LangCode aLang, LangCode bLang);
```

```
pascal short TextOrder      (const void *aPtr, const void *bPtr,
                             short aLen, short bLen,
                             ScriptCode aScript, ScriptCode bScript,
                             LangCode aLang, LangCode bLang);
```

Determining Sorting Order for Strings in the Same Language

```
pascal short RelString      (ConstStr255Param aStr, ConstStr255Param bStr,
                             Boolean caseSens, Boolean diacSens);
pascal short CompareString  (ConstStr255Param aStr, ConstStr255Param bStr,
                             Handle itl2Hande);
pascal short CompareText    (const void *aPtr, const void *bPtr,
                             short aLen, short bLen, Handle itl2Handle);
```

Modifying Characters and Diacritical Marks

```
pascal void UpperString     (Str255 theString, Boolean diacSens);
pascal void LowercaseText   (Ptr textPtr, short len, ScriptCode script);
pascal void UppercaseText   (Ptr textPtr, short len, ScriptCode script);
pascal void StripDiacritics (Ptr textPtr, short len, ScriptCode script);
pascal void UppercaseStripDiacritics
                            (Ptr textPtr, short len, ScriptCode script);
```

Truncating Strings

```
pascal short TruncString    (short width, Str255 theString,
                             TruncCode truncWhere);
pascal short TruncText      (short width, Ptr textPtr, short *textLen,
                             TruncCode truncWhere);
```

Searching for and Replacing Strings

```
pascal short ReplaceText    (Handle baseText, Handle substitutionText,
                             Str15 key);
pascal long Munger          (Handle h, long offset, const void *ptr1,
                             long len1, const void *ptr2, long len2);
```

Working With Word, Subscript, and Line Boundaries

```
pascal void FindWordBreaks  (Ptr textPtr, short textLen, short offset,
                             Boolean leadingEdge, NBreakTablePtr breaks,
                             OffsetTable offsets);
pascal StyledLineBreakCode StyledLineBreak
                            (Ptr textPtr, long textLen, long textStart,
                             long textEnd, long flags, Fixed *textWidth,
                             long *textOffset);
```

```
pascal ScriptRunStatus FindScriptRun
                            (Ptr textPtr, long textLen, long *lenUsed);
```

Converting Date and Time Strings Into Numeric Representations

```
pascal OSErr InitDateCache  (DateCachePtr theCache);
pascal StringToDateStatus StringtoDate
                            (Ptr textPtr, long textLen,
                             DateCachePtr theCache, long *lengthUsed,
                             LongDateRec *dateTime);
pascal StringToDateStatus StringToTime
                            (Ptr textPtr, long textLen,
                             DateCachePtr theCache, long *lengthUsed,
                             LongDateRec *dateTime);
```

Converting Numeric Representations Into Date and Time Strings

```
pascal void DateString      (long dateTime, DateForm longFlag,
                             Str255 result, Handle intlHandle);
pascal void TimeString      (long dateTime, Boolean wantSeconds,
                             Str255 result, Handle intlHandle);
```

Converting Long Date and Time Values Into Strings

```
pascal void LongDateString  (LongDateTime *dateTime, DateForm longFlag,
                             Str255 result, Handle intlHandle);
pascal void LongTimeString  (LongDateTime *dateTime, Boolean wantSeconds,
                             Str255 result, Handle intlHandle);
```

Converting Between Integers and Strings

```
pascal void NumToString     (long theNum, Str255 theString);
pascal void StringToNum     (ConstStr255Param theString, long *theNum);
```

Using Number Format Specification Strings for International Number Formatting

```
pascal FormatStatus StringToFormatRec
                            (ConstStr255Param inString,
                             const NumberParts *partsTable,
                             NumFormatString *outString);
pascal FormatStatus FormatRecToStr
                            (const NumFormatString *myFormatRec,
                             const NumberParts *partsTable,
                             Str255 outString, TripleInt positions);
```

Converting Between Strings and Floating-Point Numbers

```
pascal FormatStatus StringToExtended
                            (ConstStr255Param source,
                             const NumFormatString *myFormatRec,
                             const NumberParts *partsTable, extended80 *x);
pascal FormatStatus ExtendedToString
                            (extended80 x,
                             const NumFormatString *myFormatRec,
                             const NumberParts *partsTable,
                             Str255 outString);
```

Assembly-Language Summary

Trap Macros

Trap Macro Names

Pascal name	Trap macro name
`CompareText`	`_CompareText`
`DateString`	`_DateString`
`ExtendedToString`	`_ExtendedToString`
`FindScriptRun`	`_FindScriptRun`
`FindWordBreaks`	`_FindWordBreaks`
`FormatRecToString`	`_FormatRecToString`
`IdenticalText`	`_IdenticalText`
`InitDateCache`	`_InitDateCache`
`LanguageOrder`	`_LanguageOrder`
`LongDateString`	`_LongDateString`
`LongTimeString`	`_LongTimeString`
`NumToString`	`_NumToString`
`ReplaceText`	`_ReplaceText`
`ScriptOrder`	`_ScriptOrder`
`StringToDate`	`_StringToDate`
`StringToExtended`	`_StringToExtended`
`StringToFormatRec`	`_StringToFormatRec`
`StringToNum`	`_StringToNum`
`StringToTime`	`_StringToTime`
`StyledLineBreak`	`_StyledLineBreak`

Pascal name	Trap macro name
`TextOrder`	`_TextOrder`
`TimeString`	`_TimeString`
`TruncString`	`_TruncString`
`TruncText`	`_TruncText`

Trap Macros With Trap Words

Trap macro name	Trap word
`_CmpString`	`$A03C`
`_GetString`	`$A9BA`
`_LowerText`	`$A056`
`_Munger`	`$A9E0`
`_NewString`	`$A906`
`_RelString`	`$A050`
`_SetString`	`$A907`
`_StripText`	`$A256`
`_StripUpperText`	`$A656`
`_UpperText`	`$A456`
`_UprString`	`$A054`

Trap Macros Requiring Routine Selectors

`_PACK6`

Selector	Routine
$000E	`DateString`
$0010	`TimeString`
$0014	`LongDateString`
$0016	`LongTimeString`
$001A	`CompareText`
$001C	`IdenticalText`
$001E	`ScriptOrder`
$0020	`LanguageOrder`
$0022	`TextOrder`

`_PACK7`

Selector	Routine
$0000	`NumToString`
$0001	`StringToNum`

_ScriptUtil

Selector	Routine
$8204 FFF8	InitDateCache
$8208 FFE0	TruncString
$820C 0026	FindScriptRun
$820C FFDC	ReplaceText
$820C FFEC	StringToFormatRec
$820C FFDE	TruncText
$8210 FFE6	StringToExtended
$8210 FFE8	ExtendedToString
$8210 FFEA	FormatRecToString
$8214 FFF6	StringToDate
$8214 FFF4	StringToTime
$821C FFFE	StyledLineBreak
$C012 001A	FindWordBreaks