Text Utilities Reference

Carbon > Text & Fonts

2007-05-29

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

Simultaneously published in the United States and Canada.

Contents

Tables

Text Utilities Reference

Framework:	CoreServices/CoreServices.h, Carbon/Carbon.h
Declared in	NumberFormatting.h StringCompare.h TextUtils.h TypeSelect.h

Overview

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.

The Text Utilities functions 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

Carbon supports the majority of Text Utilities. However, Apple recommends that you use the comparison and word breaking utilities supplied by Unicode Utilities instead.

A number of obsolete Text Utilities functions-such as those prefixed with iu or IU-are not supported.

Functions by Task

Comparing Strings for Equality

`EqualString` (page 33) Deprecated in Mac OS X v10.4
: Compares two Pascal strings for equality, using the comparison rules of the Macintosh file system. (Deprecated. Use `CFStringCompare` instead.)

`IdenticalString` (page 41) Deprecated in Mac OS X v10.4
: Compares two Pascal strings for equality, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated. Use `CFStringCompare` instead.)

`IdenticalText` (page 41) Deprecated in Mac OS X v10.4
: Compares two text strings for equality, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated. Use `CFStringCompare` instead.)

Converting Between Integers and Strings

`NumToString` (page 46) Deprecated in Mac OS X v10.4
: Converts a long integer value into a Pascal string. (Deprecated. Use `CFStringCreateWithFormat` instead.)

`StringToNum` (page 55) Deprecated in Mac OS X v10.4
: Converts the Pascal string representation of a base-10 number into a long integer value. (Deprecated. Use `CFStringGetIntValue` instead.)

Converting Between Strings and Floating-Point Numbers

`ExtendedToString` (page 34) Deprecated in Mac OS X v10.4
: Converts an internal floating-point representation of a number into a string that can be presented to the user, using a `NumFormatStringRec` structure to specify how the output number string is formatted (Deprecated. Use `CFNumberFormatterCreateNumberFromString` instead.)

`StringToExtended` (page 53) Deprecated in Mac OS X v10.4
: Converts a string representation of a number into a floating-point number, using a `NumFormatStringRec` structure to specify how the input number string is formatted. (Deprecated. Use `CFNumberFormatterCreateStringWithNumber` instead.)

Converting Between C and Pascal Strings

`c2pstr` (page 29) Deprecated in Mac OS X v10.4
: Converts a C string to a Pascal string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`C2PStr` (page 29) Deprecated in Mac OS X v10.4
: Converts a C string to a Pascal string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`c2pstrcpy` (page 30) Deprecated in Mac OS X v10.4
Converts a C string to a Pascal string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`CopyCStringToPascal` (page 32) Deprecated in Mac OS X v10.4
Converts a C string to a Pascal string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`CopyPascalStringToC` (page 32) Deprecated in Mac OS X v10.4
Converts a Pascal String to a C string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`p2cstr` (page 46) Deprecated in Mac OS X v10.4
Converts a Pascal string to a C string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`P2CStr` (page 47) Deprecated in Mac OS X v10.4
Converts a Pascal string to a C string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

`p2cstrcpy` (page 47) Deprecated in Mac OS X v10.4
Converts a Pascal string to a C string. (Deprecated. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

Defining and Specifying Strings

`GetIndString` (page 39) Deprecated in Mac OS X v10.4
Loads a string from a string list (`'STR#'`) resource into memory, given the resource ID of the string list and the index of the individual string. (Deprecated. Use `CFBundleCopyLocalizedString` instead.)

`GetString` (page 40) Deprecated in Mac OS X v10.4
Loads a string from a string (`'STR'`) resource into memory. (Deprecated. Use `CFBundleCopyLocalizedString` instead.)

`NewString` (page 45) Deprecated in Mac OS X v10.4
Allocates memory in the heap for a string, copies its contents, and produces a handle for the heap version of the string. (Deprecated. Use `CFStringCreateCopy` instead.)

`SetString` (page 51) Deprecated in Mac OS X v10.4
Changes the contents of a string referenced by a string handle, replacing the previous contents by copying the specified string. (Deprecated. Use `CFStringCreateWithPascalString` and `CFStringReplaceAll`.)

Determining Sorting Order for Strings in Different Languages

`LanguageOrder` (page 43) Deprecated in Mac OS X v10.4
Determines the order in which strings in two different languages should be sorted. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`ScriptOrder` (page 50) Deprecated in Mac OS X v10.4
Determines the order in which strings in two different scripts should be sorted. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`StringOrder` (page 51) Deprecated in Mac OS X v10.4
Compares two Pascal strings, taking into account the script system and language for each of the strings. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`TextOrder` (page 57) Deprecated in Mac OS X v10.4
Compares two text strings, taking into account the script and language for each of the strings. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

Determining Sorting Order for Strings in the Same Language

`CompareString` (page 30) Deprecated in Mac OS X v10.4
Compares two Pascal strings, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`CompareText` (page 31) Deprecated in Mac OS X v10.4
Compares two text strings, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`RelString` (page 48) Deprecated in Mac OS X v10.4
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. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

`relstring` (page 48) Deprecated in Mac OS X v10.4
Compares two strings. (Deprecated. Use `CFStringCompare` or `UCCompareText` instead.)

Modifying Characters and Diacritical Marks

`LowercaseText` (page 44) Deprecated in Mac OS X v10.4
Converts any uppercase characters in a text string into their lowercase equivalents. (Deprecated. Use `CFStringLowercase` instead.)

`StripDiacritics` (page 56) Deprecated in Mac OS X v10.4
Strips any diacritical marks from a text string. (Deprecated. Use `CFStringTransform` instead.)

`UppercaseStripDiacritics` (page 61) Deprecated in Mac OS X v10.4
Converts any lowercase characters in a text string into their uppercase equivalents and strips any diacritical marks from the text. (Deprecated. Use `CFStringTransform` instead.)

`UppercaseText` (page 62) Deprecated in Mac OS X v10.4
Converts any lowercase characters in a text string into their uppercase equivalents. (Deprecated. Use `CFStringUppercase` instead.)

`UpperString` (page 63) Deprecated in Mac OS X v10.4
Converts any lowercase letters in a Pascal string to their uppercase equivalents, using the Macintosh file system rules. (Deprecated. Use `CFStringUppercase` instead.)

`upperstring` (page 64) Deprecated in Mac OS X v10.4
Converts any lowercase letters in a Pascal string to their uppercase equivalents. (Deprecated. Use `CFStringUppercase` instead.)

Searching for and Replacing Strings

`Munger` (page 14)
Searches text for a specified string pattern and replaces it with another string.

`ReplaceText` (page 49) Deprecated in Mac OS X v10.4
Searches text on a character-by-character basis, replacing all instances of a string in that text with another string. (Deprecated. Use `CFStringReplace` instead.)

Using Number Format Specification Strings for International Number Formatting

`FormatRecToString` (page 38) Deprecated in Mac OS X v10.4
Converts an internal representation of number formatting information into a number format specification string, which can be displayed and modified. (Deprecated. Use `CFNumberFormatterGetFormat` instead.)

`StringToFormatRec` (page 54) Deprecated in Mac OS X v10.4
Creates a number format specification string structure from a number format specification string that you supply in a Pascal string. (Deprecated. Use `CFNumberFormatterSetFormat` instead.)

Working With Word, Script, and Line Boundaries

`FindScriptRun` (page 35) Deprecated in Mac OS X v10.4
Finds the next block of subscript text within a script run. (Deprecated. There is no replacement function because this capability is no longer needed in Mac OS X.)

`FindWordBreaks` (page 36) Deprecated in Mac OS X v10.4
Determines the beginning and ending boundaries of a word in a text string. (Deprecated. Use `UCFindTextBreak` instead.)

Working With Universal Procedure Pointers

`DisposeIndexToStringUPP` (page 33) Deprecated in Mac OS X v10.4
Disposes of a universal procedure pointer to an index-to-string callback.

`InvokeIndexToStringUPP` (page 42) Deprecated in Mac OS X v10.4
Call an index-to-string callback.

`NewIndexToStringUPP` (page 44) Deprecated in Mac OS X v10.4
Creates a new universal procedure pointer (UPP) to an index-to-string callback.

Working With Type Select Records

`TypeSelectClear` (page 58) Deprecated in Mac OS X v10.4
Clears the key list and resets the type select record. (Deprecated. Use `UCTypeSelectFlushSelectorData` instead.)

`TypeSelectCompare` (page 59) Deprecated in Mac OS X v10.4
Compares a text buffer to the keystroke buffer. (Deprecated. Use `UCTypeSelectCompare` instead.)

`TypeSelectFindItem` (page 60) Deprecated in Mac OS X v10.4
Finds the closest match between a specified list of characters and the keystrokes stored in the type select record. (Deprecated. Use `UCTypeSelectFindItem` instead.)

`TypeSelectNewKey` (page 61) Deprecated in Mac OS X v10.4
Creates a new type select record. (Deprecated. Use `UCTypeSelectCreateSelector` instead.)

Functions

Munger

Searches text for a specified string pattern and replaces it with another string.

```
long Munger (
   Handle h,
   long offset,
   const void *ptr1,
   long len1,
   const void *ptr2,
   long len2
);
```

Parameters

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.

Return Value

A negative value if `Munger` cannot find the designated string.

Discussion

`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`.

`Munger` operates on a byte-by-byte basis, which can produce inappropriate results for 2-byte script systems. The `ReplaceText` (page 49) 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 `NULL` or if either of the length values is `0`.

- If `ptr1` is `NULL`, `Munger` replaces characters without searching. It replaces `len1` characters starting at the `offset` location with the substitution string.

- If `ptr1` is `NULL` 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 `NULL`, `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 `NULL`, `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.

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

`Munger` calls the `GetHandleSize` and `SetHandleSize` functions to access or modify the length of the string it is manipulating.

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.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`TextUtils.h`

Callbacks

IndexToStringProcPtr

Defines a pointer to your index-to-string callback function that retrieves the string associated with an index value.

Not recommended.

```
typedef Boolean (*IndexToStringProcPtr)
(
    short item,
    ScriptCode * itemsScript,
    StringPtr * itemsStringPtr,
    void * yourDataPtr
);
```

If you name your function `MyIndexToStringProc`, you would declare it like this:

```
Boolean MyIndexToStringProcPtr (
    short item,
    ScriptCode * itemsScript,
```

```
    StringPtr * itemsStringPtr,
    void * yourDataPtr
);
```

Parameters

item

The index value for which the `TypeSelect` function requests a string.

itemsScript

The script code of the string specified by `itemsStringPtr`.

itemsStringPtr

On return, points to the string that matches the index specify by the `item` parameter.

yourDataPtr

A pointer to your data structure. This is passed to your index-to-string callback, and can be `NULL`, depending on how you implement your callback function.

Return Value

Returns `true` if a string matching that index value was found; `false` otherwise.

Discussion

The use of this function is not recommended in a Unicode-based application. If you want to use this function in an application that uses the Unicode character set, you must first convert Unicode text strings to Macintosh encoded Pascal text strings. You must also provide the encoding type or be able to determine it by extracting it from the text or by examining the system or keyboard script.

Availability

Not recommended. Available in CarbonLib 1.0 and later.

Available in Mac OS X 10.0 and later.

Not available to 64-bit applications.

Declared In

`TypeSelect.h`

Data Types

BreakTable

Contains information used to determine the boundaries of a word.

```
struct BreakTable {
    char charTypes[256];
    short tripleLength;
    short triples[1];
};
typedef struct BreakTable BreakTable;
typedef BreakTable * BreakTablePtr;
```

Discussion

You can supply a `BreakTable` as a parameter to the function `FindWordBreaks` (page 36).

Availability
Available in Mac OS X v10.0 and later.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

FormatClass

Defines a data type used to access entries in a triple integer array.

```
typedef SInt8 FormatClass;
```

Discussion
Each of the three `FVector` entries in a triple integer array is accessed by one of the values of the `FormatClass` type. See `FVector` (page 17) for more information.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

FormatStatus

Defines a data type used to denote the confidence level for a conversion.

```
typedef short FormatStatus;
```

Discussion
A `FormatStatus` value is returned by the functions `ExtendedToString` (page 34), `StringToExtended` (page 53), `FormatRecToString` (page 38), and `StringToFormatRec` (page 54).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

FVector

Contains position and length information for one portion of a formatted numeric string.

```
struct FVector {
    short start;
    short length;
};
typedef struct FVector FVector;
typedef FVector TripleInt[3];
```

Fields
`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.

Discussion
The `FVector` data structure is used in the `TripleInt` (page 22) array.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

IndexToStringUPP

Defines a universal procedure pointer to an index-to-string callback.

```
typedef IndexToStringProcPtr IndexToStringUPP;
```

Discussion
For more information, see the description of the `IndexToStringProcPtr` (page 15) callback function.

Availability
Available in Mac OS X v10.0 and later.
Not available to 64-bit applications.

Declared In
`TypeSelect.h`

NBreakTable

Contains information used by the `FindWordBreaks` function to determine word boundaries.

```
struct NBreakTable {
    SInt8 flags1;
    SInt8 flags2;
    short version;
    short classTableOff;
    short auxCTableOff;
    short backwdTableOff;
    short forwdTableOff;
    short doBackup;
    short length;
    char charTypes[256];
    short tables[1];
};
typedef struct NBreakTable NBreakTable;
typedef NBreakTable * NBreakTablePtr;
```

Fields
`flags1`
The high-order byte of the break table format flags. If the high-order bit of this byte is set to 1, this break table is in the format used by `FindWordBreaks`.

`flags2`
: The low-order byte of the break table format flags. If the value in this byte is 0, the break table belongs to a 1-byte script system; in this case `FindWordBreaks` does not check for 2-byte characters.

`version`
: The version of this break table.

`classTableOff`
: The offset in bytes from the beginning of the break table to the beginning of the class table.

`auxCTableOff`
: The offset in bytes from the beginning of the break table to the beginning of the auxiliary class table.

`backwdTableOff`
: The offset in bytes from the beginning of the break table to the beginning of the backward-processing table.

`forwdTableOff`
: The offset in bytes from the beginning of the break table to the beginning of the forward-processing table.

`doBackup`
: The minimum byte offset into the buffer for doing backward processing. If the selected character for `FindWordBreaks` has a byte offset less than `doBackup`, `FindWordBreaks` skips backward processing altogether and starts from the beginning of the buffer.

`length`
: The length in bytes of the entire break table, including the individuals tables.

`charTypes`
: The class table.

`tables`
: The data of the auxiliary class table, backward table, and forward table.

Discussion
The tables have this format and content:

- The class table is an array of 256 signed bytes. Offsets into the table represent byte values; if the entry at a given offset in the table is positive, it means that a byte whose value equals that offset is a single-byte character, and the entry at that offset is the class number for the character. If the entry is negative, it means that the byte is the first byte of a 2-byte character code, and the auxiliary class table must be used to determine the character class. Odd negative numbers are handled differently from even negative numbers.
- The auxiliary class table assigns character classes to 2-byte characters. It is used when the class table determines that a byte value represents the first byte of a 2-byte character.
  - The auxiliary class table handles odd negative values from the class table as follows. If the first word of the auxiliary class table is equal to or greater than zero, it represents the default class number for 2-byte character codes—the class assigned to every odd negative value from the class table. If the first word is less than zero, it is the negative of the offset from the beginning of the auxiliary class table to a first-byte class table (a table of 2-byte character classes that can be determined from just the first byte). The value from the class table is negated, 1 is subtracted from it to obtain an even offset, and the value at that offset into the first-byte class table is the class to be assigned.
  - The auxiliary class table handles even negative values from the class table as follows. The auxiliary class table begins with a variable-length word array. The words that follow the first word are offsets to row tables. Row tables have the same format as the class table, but are used to map the second byte of a 2-byte character code to a class number. If the entry in the class table for a given byte is an even negative number, `FindWordBreaks` negates this value to obtain the offset from the

beginning of the auxiliary class table to the appropriate word, which in turn contains an offset to the appropriate row table. That row table is then used to map the second byte of the character to a class number.

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

Availability
Available in Mac OS X v10.0 and later.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

NumFormatString

Contains data that represents the internal number formatting specification.

```
struct NumFormatString {
    UInt8 fLength;
    UInt8 fVersion;
    char data[254];
};
typedef struct NumFormatString NumFormatString;
typedef NumFormatString NumFormatStringRec;
```

Fields

`fLength`
The number of bytes in the data actually used for this number formatting specification.

`fVersion`
The version number of the number formatting specification.

`data`
The data that comprises the number formatting specification.

Discussion

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

NumFormatStringRec

Defines an internal numeric representation that is independent of region, language, and other multicultural consideration.

```
typedef NumFormatString NumFormatStringRec;
```

Discussion
To allow for all of the international variations in numeric presentation styles, you need to include in your function 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.

To make it possible to map a number that was formatted for one specification into another format, the Mac OS defines an internal numeric representation that is independent of region, language, and other multicultural considerations: the `NumFormatStringRec` structure. This structure is created from a number format specification string that defines the appearance of numeric strings.

Four of the numeric string functions use the number formatting specification, defined by the `NumFormatStringRec` data type: `StringToFormatRec` (page 54), `FormatRecToString` (page 38), `StringToExtended` (page 53), and `ExtendedToString` (page 34). The number format specification structure contains the data that represents the internal number formatting specification information. This data is stored in a private format.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

ScriptRunStatus

Contains script-specific information for a script run.

```
struct ScriptRunStatus {
    SInt8 script;
    SInt8 runVariant;
};
typedef struct ScriptRunStatus ScriptRunStatus;
```

Fields

`script`
The script code of the subscript run. Zero indicates the Roman script system.

`runVariant`
Script-specific information about the run, in the same format as that returned by the `CharacterType` function. This information includes the type of subscript—for example, Kanji, Katakana, or Hiragana for a Japanese script system.

Discussion
The `FindScriptRun` (page 35) function returns the script run status structure, defined by the `ScriptRunStatus` data type, when it completes its processing, which is to find a run of subscript text in a string.

Availability
Available in Mac OS X v10.0 and later.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

TripleInt

Defines a data type used to return the position and length information for three different portions of a formatted numeric string.

```
typedef FVector TripleInt[3];
```

Discussion
The `FormatRecToString` (page 38) 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` structure. Each of the three `FVector` entries in the triple integer array is accessed by one of the values of the `FormatClass` type.

Availability
Available in Mac OS X v10.0 and later.

Declared In
`NumberFormatting.h`

TypeSelectRecord

Contains a buffer of keystrokes, the script code associated with the keystrokes, and timer information.

```
struct TypeSelectRecord {
    unsigned long tsrLastKeyTime;
    ScriptCode tsrScript;
    Str63 tsrKeyStrokes;
};
typedef struct TypeSelectRecord TypeSelectRecord;
```

Fields

`tsrLastKeyTime`
A value that indicates timeout information.

`tsrScript`
A script code.

`tsrKeyStrokes`
The keystroke buffer.

Discussion
The `TypeSelectRecord` data structure is passed as a parameter to the functions `TypeSelectNewKey` (page 61), `TypeSelectFindItem` (page 60), `TypeSelectCompare` (page 59), and `TypeSelectClear` (page 58).

Availability
Available in Mac OS X v10.0 and later.

Declared In
`TypeSelect.h`

Constants

Format Result Types

Specify values that can be returned in the low byte of a format status (`FormatStatus`) value.

```
enum {
    fFormatOK = 0,
    fBestGuess = 1,
    fOutOfSynch = 2,
    fSpuriousChars = 3,
    fMissingDelimiter = 4,
    fExtraDecimal = 5,
    fMissingLiteral = 6,
    fExtraExp = 7,
    fFormatOverflow = 8,
    fFormStrIsNAN = 9,
    fBadPartsTable = 10,
    fExtraPercent = 11,
    fExtraSeparator = 12,
    fEmptyFormatString = 13
};
typedef SInt8 FormatResultType;
```

Constants

`fFormatOK`

Specifies format is okay.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fBestGuess`

Specifies the format is the best guess.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fOutOfSynch`

Specifies the format is out of sync.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fSpuriousChars`

Specifies the format contains spurious characters.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fMissingDelimiter`

Specifies a missing delimiter.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fExtraDecimal`
Specifies the format contains an extra decimal sign.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fMissingLiteral`
Specifies the format is missing a literal.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fExtraExp`
Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fFormatOverflow`
Specifies a format overflow.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fFormStrIsNAN`
Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fBadPartsTable`
Specifies the parts table is bad.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fExtraPercent`
Specifies the format contains an extra percent sign.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fExtraSeparator`
Specifies an extra separator.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fEmptyFormatString`
Specifies the format string is empty.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

Discussion
A format result type is returned in the low byte of a format status (`FormatStatus`) value. A `FormatStatus` (page 17) value is returned by the functions `ExtendedToString` (page 34), `StringToExtended` (page 53), `FormatRecToString` (page 38), and `StringToFormatRec` (page 54). A format status value denotes the confidence level for a conversion.

TripleInt Index Values

Specify an index for a `TripleInt` array.

```
enum {
    fPositive = 0,
    fNegative = 1,
    fZero = 2
};
```

Constants

`fPositive`

Specifies the positive value string.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fNegative`

Specifies the negative value string.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

`fZero`

Specifies the zero value string.

Available in Mac OS X v10.0 and later.

Declared in `NumberFormatting.h`.

Discussion

See `TripleInt` (page 22) for more information.

NumFormatString Version

Specifies the first version of the `NumFormatString` data structure.

```
enum {
    fVNumber = 0
};
```

Discussion

See `NumFormatString` (page 20) for more information.

Implicit Language Codes

Specify implicit language codes.

```
enum {
    systemCurLang = -2,
    systemDefLang = -3,
    currentCurLang = -4,
    currentDefLang = -5,
    scriptCurLang = -6,
    scriptDefLang = -7
};
```

Constants

`systemCurLang`
Specifies the current language for system script (from `'itlb'`).
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

`systemDefLang`
Specifies the default language for system script (from `'itlm'`).
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

`currentCurLang`
Specifies the current language for current script (from `'itlb'`).
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

`currentDefLang`
Specifies the default language for current script (from `'itlm'`).
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

`scriptCurLang`
Specifies the current language for specified script (from `'itlb'`)
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

`scriptDefLang`
Specifies the default language for specified script (from `'itlm'`)
Available in Mac OS X v10.0 and later.
Declared in `StringCompare.h`.

Discussion

The functions `LanguageOrder` (page 43), `StringOrder` (page 51), and `TextOrder` (page 57) accept as parameters implicit language codes listed here, as well as explicit language codes.

Type Select Modes

Contains type-select code information.

```
typedef SInt16 TSCode;
enum {
    tsPreviousSelectMode = -1,
    tsNormalSelectMode = 0,
    tsNextSelectMode = 1
};
```

Constants

`tsPreviousSelectMode`
Specifies previous-select mode.
Available in Mac OS X v10.0 and later.
Declared in `TypeSelect.h`.

`tsNormalSelectMode`
Specifies normal-select mode.
Available in Mac OS X v10.0 and later.
Declared in `TypeSelect.h`.

`tsNextSelectMode`
Specifies next-select mode.
Available in Mac OS X v10.0 and later.
Declared in `TypeSelect.h`.

Discussion
This structure is passed as a parameter to the function `TypeSelectFindItem` (page 60).

Obsolete Language Code Values

Specify language code values that are no longer used.

```
enum {
    iuSystemCurLang = systemCurLang,
    iuSystemDefLang = systemDefLang,
    iuCurrentCurLang = currentCurLang,
    iuCurrentDefLang = currentDefLang,
    iuScriptCurLang = scriptCurLang,
    iuScriptDefLang = scriptDefLang
};
```

Deprecated Text Utilities Functions

A function identified as deprecated has been superseded and may become unsupported in the future.

Deprecated in Mac OS X v10.4

c2pstr

Converts a C string to a Pascal string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
StringPtr c2pstr (
   char *aStr
);
```

Availability
Available in Mac OS X v10.4 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

C2PStr

Converts a C string to a Pascal string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
StringPtr C2PStr (
    Ptr cString
);
```

Availability
Available in Mac OS X v10.4 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

c2pstrcpy

Converts a C string to a Pascal string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
void c2pstrcpy (
   Str255 dst,
   const char *src
);
```

Parameters

dst

On output, the Pascal string.

src

The C string you want to convert.

Discussion

This function allows in-place conversion. That is, the `src` and `dst` parameters can point to the same memory location.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Related Sample Code

SoftVDigX

Declared In

`TextUtils.h`

CompareString

Compares two Pascal strings, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short CompareString (
    ConstStr255Param aStr,
    ConstStr255Param bStr,
    Handle itl2Handle
);
```

Parameters

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. If the value of this parameter is `NULL`, `CompareString` makes use of the resource for the current script. The string-manipulation resource includes functions and tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Return Value
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.

Discussion
This function 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.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

CompareText

Compares two text strings, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short CompareText (
   const void *aPtr,
   const void *bPtr,
   short aLen,
   short bLen,
   Handle itl2Handle
);
```

Parameters

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 length, in bytes, of the first text string.

bLen
The length, in bytes, of the second text string.

itl2Handle
A handle to a string-manipulation (`'itl2'`) resource that contains string comparison information. If the value of this parameter is `NULL`, `CompareText` makes use of the resource for the current script. The string-manipulation resource includes functions and tables for modifying string comparison and tables for case conversion and stripping of diacritical marks.

Return Value
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.

Discussion
This function 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.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

CopyCStringToPascal

Converts a C string to a Pascal string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
void CopyCStringToPascal (
   const char *src,
   Str255 dst
);
```

Parameters

src
The C string you want to convert.

dst
On output, the Pascal string.

Discussion
This function allows in-place conversion. That is, the `src` and `dst` parameters can point to the same memory location.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Related Sample Code
BSDLLCTest

Declared In
`TextUtils.h`

CopyPascalStringToC

Converts a Pascal String to a C string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
void CopyPascalStringToC (
   ConstStr255Param src,
   char *dst
);
```

Parameters

src

The Pascal string you want to convert.

dst

On output, the C string.

Discussion

This function allows in-place conversion. That is, the `src` and `dst` parameters can point to the same memory location.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

DisposeIndexToStringUPP

Disposes of a universal procedure pointer to an index-to-string callback. (Deprecated in Mac OS X v10.4.)

```
void DisposeIndexToStringUPP (
   IndexToStringUPP userUPP
);
```

Parameters

userUPP

The universal procedure pointer.

Discussion

See the callback `IndexToStringProcPtr` (page 15) for more information.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TypeSelect.h`

EqualString

Compares two Pascal strings for equality, using the comparison rules of the Macintosh file system. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` instead.)

```
Boolean EqualString (
   ConstStr255Param str1,
   ConstStr255Param str2,
   Boolean caseSensitive,
   Boolean diacSensitive
);
```

Parameters

str1

One of the Pascal strings to be compared.

str2

The other Pascal string to be compared.

caseSensitive

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.

diacSensitive

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.

Return Value

`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.

Discussion

The comparison is a simple, character-by-character value comparison. This function does not make use of any script or language information (i.e., is not localizable); it assumes the use of a Roman script system.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`StringCompare.h`

ExtendedToString

Converts an internal floating-point representation of a number into a string that can be presented to the user, using a `NumFormatStringRec` structure to specify how the output number string is formatted (Deprecated in Mac OS X v10.4. Use `CFNumberFormatterCreateNumberFromString` instead.)

```
FormatStatus ExtendedToString (
   const extended80 *x,
   const NumFormatString *myCanonical,
   const NumberParts *partsTable,
   Str255 outString
);
```

Parameters

x

A pointer to a floating-point value in 80-bit SANE representation.

myCanonical

A pointer to the internal representation of the formatting information for numbers, as produced by the `StringToFormatRec` function.

partsTable

A pointer to a structure, 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`.

Return Value

A value that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`. Be sure to cast the result of `ExtendedToString` to a type `FormatResultType` before working with it. See the description of the `FormatStatus` data type.

Discussion

`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.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` function.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`NumberFormatting.h`

FindScriptRun

Finds the next block of subscript text within a script run. (Deprecated in Mac OS X v10.4. There is no replacement function because this capability is no longer needed in Mac OS X.)

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

Parameters

textPtr

A pointer to the text string to be analyzed.

textLen

The number of bytes in the text string.

lenUsed

On output, a pointer to 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.

Return Value

Identifies the run as either native text, Roman, or one of the defined subscripts of the script system and returns a structure of type `ScriptRunStatus` (page 21). See the description of the `ScriptRunStatus` data type.

Discussion

The `FindScriptRun` function is used to identify blocks of subscript text in a string, taking into account script and language considerations, making use of tables in the string-manipulation (`'itl2'`) resource in its computations. 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.

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.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

FindWordBreaks

Determines the beginning and ending boundaries of a word in a text string. (Deprecated in Mac OS X v10.4. Use `UCFindTextBreak` instead.)

```
void FindWordBreaks (
   Ptr textPtr,
   short textLength,
   short offset,
   Boolean leadingEdge,
   BreakTablePtr breaks,
   OffsetTable offsets,
   ScriptCode script
);
```

Parameters

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 preceding the offset.

breaks

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.

Discussion

`FindWordBreaks` searches for a word in a text string, taking into account script and language considerations, making use of tables in the string-manipulation (`'itl2'`) resource in its computations. 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.

`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.

`FindWordBreaks` returns its results in an `OffsetTable` structure. `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`).

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

FormatRecToString

Converts an internal representation of number formatting information into a number format specification string, which can be displayed and modified. (Deprecated in Mac OS X v10.4. Use `CFNumberFormatterGetFormat` instead.)

```
FormatStatus FormatRecToString (
   const NumFormatString *myCanonical,
   const NumberParts *partsTable,
   Str255 outString,
   TripleInt positions
);
```

Parameters

myCanonical
A pointer to the internal representation of number formatting information, as created by a previous call to the `StringToFormatRec` function.

partsTable
A pointer to a structure, 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.

Return Value
A value that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`. Be sure to cast the result of `FormatRecToString` to a type `FormatResultType` before working with it. See the description of the `FormatStatus` data type.

Discussion
`FormatRecToString` is the inverse operation of `StringToFormatRec` (page 54). 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 `positions` parameter is an array of three integers (a `TripleInt` value), which specifies the starting position in `outString` of each of three formatting specifications:

- `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.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` function.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`NumberFormatting.h`

GetIndString

Loads a string from a string list (`'STR#'`) resource into memory, given the resource ID of the string list and the index of the individual string. (Deprecated in Mac OS X v10.4. Use `CFBundleCopyLocalizedString` instead.)

```
void GetIndString (
   Str255 theString,
   short strListID,
   short index
);
```

Parameters

`theString`
On output, the Pascal string result; specifically, 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.

`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.

Discussion
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`.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

GetString

Loads a string from a string (`'STR'`) resource into memory. (Deprecated in Mac OS X v10.4. Use `CFBundleCopyLocalizedString` instead.)

```
StringHandle GetString (
   short stringID
);
```

Parameters

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

Return Value
A handle to a string with the specified resource ID. If necessary, `GetString` reads the handle form the resource file. If `GetString` cannot read the resource, it returns `NULL`.

Discussion
`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` (page 45) function, `GetString` returns a handle whose size is based upon the actual length of the string.

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` (page 39) function 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.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
TextUtils.h

IdenticalString

Compares two Pascal strings for equality, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` instead.)

```
short IdenticalString (
   ConstStr255Param aStr,
   ConstStr255Param bStr,
   Handle itl2Handle
);
```

Parameters

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.

The `itl2Handle` parameter is used to specify a string-manipulation resource. If the value of this parameter is `NULL`, `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.

Return Value
Returns 0 if the two strings are equal; 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.

Discussion
`IdenticalString` uses only primary differences in its comparison.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
StringCompare.h

IdenticalText

Compares two text strings for equality, making use of the string comparison information from a resource that you specify as a parameter. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` instead.)

```
short IdenticalText (
   const void *aPtr,
   const void *bPtr,
   short aLen,
   short bLen,
   Handle itl2Handle
);
```

Parameters

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 length, in bytes, of the first text string.

bLen

The length, in bytes, of the second text string.

itl2Handle

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

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

Return Value

0 if the two text strings are equal; 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.

Discussion

`IdenticalText` uses only primary sorting order in its comparison.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`StringCompare.h`

InvokeIndexToStringUPP

Call an index-to-string callback. (Deprecated in Mac OS X v10.4.)

```
Boolean InvokeIndexToStringUPP (
   short item,
   ScriptCode *itemsScript,
   StringPtr *itemsStringPtr,
   void *yourDataPtr,
   IndexToStringUPP userUPP
);
```

Discussion
You should not need to use the function `InvokeIndexToStringUPP`, as the system calls your index-to-string callback function for you. See the callback `IndexToStringProcPtr` (page 15) for more information.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TypeSelect.h`

LanguageOrder

Determines the order in which strings in two different languages should be sorted. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short LanguageOrder (
   LangCode language1,
   LangCode language2
);
```

Parameters

language1
The language code of the first language.

language2
The language code of the second language.

Return Value
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).

Discussion
`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.

"Implicit Language Codes" (page 25) are listed in the Constants section. 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.

Availability
Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

LowercaseText

Converts any uppercase characters in a text string into their lowercase equivalents. (Deprecated in Mac OS X v10.4. Use `CFStringLowercase` instead.)

```
void LowercaseText (
   Ptr textPtr,
   short len,
   ScriptCode script
);
```

Parameters

`textPtr`

A pointer to the text string to be converted.

`len`

The number of bytes in the text string. The text string can be up to 32 KB in length.

`script`

The script code for the script system whose resources are used to determine the results of converting characters.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

Discussion
`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.

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.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

NewIndexToStringUPP

Creates a new universal procedure pointer (UPP) to an index-to-string callback. (Deprecated in Mac OS X v10.4.)

```
IndexToStringUPP NewIndexToStringUPP (
   IndexToStringProcPtr userRoutine
);
```

Parameters

userRoutine

A pointer to your index-to-string callback.

Return Value

On return, a UPP to the index-to-string callback.

Discussion

See the callback `IndexToStringProcPtr` (page 15) for more information.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TypeSelect.h`

NewString

Allocates memory in the heap for a string, copies its contents, and produces a handle for the heap version of the string. (Deprecated in Mac OS X v10.4. Use `CFStringCreateCopy` instead.)

```
StringHandle NewString (
   ConstStr255Param theString
);
```

Parameters

theString

A Pascal string that you want copied onto the heap.

Return Value

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

Discussion

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` (page 51) function or the Memory Manager function `SetHandleSize` to modify the string's size.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

NumToString

Converts a long integer value into a Pascal string. (Deprecated in Mac OS X v10.4. Use `CFStringCreateWithFormat` instead.)

```
void NumToString (
   long theNum,
   Str255 theString
);
```

Parameters

`theNum`

A long integer value. If the value of the number in the parameter `theNum` is negative, the string begins with a minus sign; otherwise, the sign is omitted.

`theString`

On output, contains the Pascal string representation of the number. 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.

Discussion

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

Unless patched by a script system with different rules, this function assumes that you are using standard numeric token processing, meaning that the Roman script system number processing rules are used.

For functions that make use of the token-processing information that is found in the tokens (`'itl4'`) resource of script systems for converting numbers, see the sections "Using Number Format Specification Strings for International Number Formatting" and "Converting Between Strings and Floating-Point Numbers".

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`NumberFormatting.h`

p2cstr

Converts a Pascal string to a C string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
char * p2cstr (
   StringPtr aStr
);
```

Availability

Available in Mac OS X v10.4 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In
`TextUtils.h`

P2CStr

Converts a Pascal string to a C string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
Ptr P2CStr (
    StringPtr pString
);
```

Availability
Available in Mac OS X v10.4 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

p2cstrcpy

Converts a Pascal string to a C string. (Deprecated in Mac OS X v10.4. You should store strings as Core Foundation CFStrings instead. See *CFString Reference*.)

```
void p2cstrcpy (
   char *dst,
   ConstStr255Param src
);
```

Parameters

dst
On output, the C string.

src
The Pascal string you want to convert.

Discussion
This function allows in-place conversion. That is, the `src` and `dst` parameters can point to the same memory location.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

RelString

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. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short RelString (
   ConstStr255Param str1,
   ConstStr255Param str2,
   Boolean caseSensitive,
   Boolean diacSensitive
);
```

Parameters

str1

One of the Pascal strings to be compared.

str2

The other Pascal string to be compared.

caseSensitive

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.

diacSensitive

A flag that indicates how to handle information about diacritical marks during the string comparison. If the value of `diacSensitive` 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.

Return Value

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.

Discussion

This function does not make use of any script or language information; it assumes the original Macintosh character set only.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`StringCompare.h`

relstring

Compares two strings. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

Not recommended

```
short relstring (
   const char *str1,
   const char *str2,
   Boolean caseSensitive,
   Boolean diacSensitive
);
```

Parameters

str1

The string to be compared to `str2`.

str2

The string to be compared to `str1`.

caseSensitive

A flag that indicates how to handle case-sensitive information during the comparison.

diacSensitive

A flag that indicates how to handle information about diacritical marks during the string comparison.

Return Value

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.

Discussion

This function is not recommended. Instead, see the function `RelString` (page 48).

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`StringCompare.h`

ReplaceText

Searches text on a character-by-character basis, replacing all instances of a string in that text with another string. (Deprecated in Mac OS X v10.4. Use `CFStringReplace` instead.)

```
short ReplaceText (
    Handle baseText,
    Handle substitutionText,
    Str15 key
);
```

Parameters

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.

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

Discussion
`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 (as opposed to byte-by-byte), so it works properly for all script systems, including 2-byte script systems. It recognizes 2-byte characters in script systems that contain them and advances the search appropriately after encountering a 2-byte character.

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

ScriptOrder

Determines the order in which strings in two different scripts should be sorted. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short ScriptOrder (
   ScriptCode script1,
   ScriptCode script2
);
```

Parameters

script1
The script code of the first script.

script2
The script code of the second script.

Return Value
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).

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

Special Considerations

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

Availability
Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

SetString

Changes the contents of a string referenced by a string handle, replacing the previous contents by copying the specified string. (Deprecated in Mac OS X v10.4. Use `CFStringCreateWithPascalString` and `CFStringReplaceAll`.)

```
void SetString (
   StringHandle theString,
   ConstStr255Param strNew
);
```

Parameters

`theString`
A Pascal string.

`strNew`
A handle to the string in memory whose contents you are replacing. If the new string (`theString`) is larger than the string originally referenced by `strNew`, `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 function at interrupt time.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TextUtils.h`

StringOrder

Compares two Pascal strings, taking into account the script system and language for each of the strings. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

```
short StringOrder (
   ConstStr255Param aStr,
   ConstStr255Param bStr,
   ScriptCode aScript,
   ScriptCode bScript,
   LangCode aLang,
   LangCode bLang
);
```

Parameters

aStr

One of the Pascal strings to be compared.

bStr

The other Pascal string to be compared.

aScript

The script code for the second string.

bScript

The script code for the first string.

aLang

The language code for the first string.

bLang

The language code for the second string.

Return Value

–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.

Discussion

This function 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.

"Implicit Language Codes" (page 25) are listed in the Constants section. Most applications specify the language code `scriptCurLang` for both the `aLang` and `bLang` values.

`StringOrder` first calls `ScriptOrder` (page 50); 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` (page 43); 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, it uses the sorting rules specified by the system script or the font script, depending on the state of the international resources selection flag.

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` (page 30) function, 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.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

StringToExtended

Converts a string representation of a number into a floating-point number, using a `NumFormatStringRec` structure to specify how the input number string is formatted. (Deprecated in Mac OS X v10.4. Use `CFNumberFormatterCreateStringWithNumber` instead.)

```
FormatStatus StringToExtended (
   ConstStr255Param source,
   const NumFormatString *myCanonical,
   const NumberParts *partsTable,
   extended80 *x
);
```

Parameters

source
A Pascal string that contains the string representation of a number.

myCanonical
A pointer to the internal representation of the formatting information for numbers, as produced by the `StringToFormatRec` function.

partsTable
A pointer to a structure, 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 a pointer to the 80-bit SANE representation of the floating-point number.

Return Value
A value that denotes the confidence level for the conversion that it performed. The low byte of the `FormatStatus` value is of type `FormatResultType`. Be sure to cast the result of `StringToExtended` to a type `FormatResultType` before working with it. `StringToExtended` returns an 80-bit, not a 96-bit, representation. See the description of the `FormatStatus` data type.

Discussion
`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.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` function.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`NumberFormatting.h`

StringToFormatRec

Creates a number format specification string structure from a number format specification string that you supply in a Pascal string. (Deprecated in Mac OS X v10.4. Use `CFNumberFormatterSetFormat` instead.)

```
FormatStatus StringToFormatRec (
   ConstStr255Param inString,
   const NumberParts *partsTable,
   NumFormatString *outString
);
```

Parameters

inString

A Pascal string that contains the number formatting specification.

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.

partsTable

A pointer to a structure, 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, a pointer to a `NumFormatStringRec` structure that contains the values that form the internal representation of the format specification. The format of the data in this structure is private.

Return Value
A value that denotes the confidence level for the conversion that was performed. The low byte of the value is of type `FormatResultType`. Be sure to cast the result of `StringToFormatRec` to a type `FormatResultType` before working with it. See the description of the `FormatStatus` data type.

Discussion
`StringToFormatRec` converts a number format specification string into the internal representation contained in a number format string structure. 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.

To obtain a handle to the number parts table from a tokens resource, use the `GetIntlResourceTable` function.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`NumberFormatting.h`

StringToNum

Converts the Pascal string representation of a base-10 number into a long integer value. (Deprecated in Mac OS X v10.4. Use `CFStringGetIntValue` instead.)

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

Parameters

`theString`

A Pascal string representation of a base-10 number. The numeric string can be padded with leading zeros or with a sign.

`theNum`

On output, contains a pointer to the numeric value.

Discussion

Unless patched by a script system with different rules, this function assumes that you are using standard numeric token processing, meaning that the Roman script system number processing rules are used.

For functions that make use of the token-processing information that is found in the tokens (`'itl4'`) resource of script systems for converting numbers, see the sections "Using Number Format Specification Strings for International Number Formatting" and "Converting Between Strings and Floating-Point Numbers".

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`:

- The value of `theString` is "-23". `StringToNum` returns the value -23 in `theNum`.
- The value of `theString` is "-0". `StringToNum` returns the value 0 in `theNum`.
- The value of `theString` is "055". `StringToNum` returns the value 55 in `theNum`.
- The value of `theString` is "2147483648" (magnitude is 2^31). `StringToNum` returns the value –2147483648 in `theNum`.
- The value of `theString` is "–2147483648". `StringToNum` returns the value –2147483648 in `theNum`.
- The value of `theString` is "4294967295" (magnitude is 2^32–1). `StringToNum` returns the value -1 in `theNum`.
- The value of `theString` is "–4294967295". `StringToNum` returns the value 1 in `theNum`.

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. Hence, the strings 'CAT', '+CAT', and '-CAT' would produce the results 314, 314, and -314.

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 function at interrupt time.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
NumberFormatting.h

StripDiacritics

Strips any diacritical marks from a text string. (Deprecated in Mac OS X v10.4. Use CFStringTransform instead.)

```
void StripDiacritics (
   Ptr textPtr,
   short len,
   ScriptCode script
);
```

Parameters

textPtr
A pointer to the text string to be stripped.

len
The length, in bytes, of the text string. The text string can be up to 32 KB in length.

script
The script code for the script system whose rules are used for determining which character results from stripping a diacritical mark.

The conversion uses tables in the string-manipulation ('itl2') resource of the script specified by the value of the script parameter. You can specify smSystemScript to use the system script and smCurrentScript to use the script of the current font in the current graphics port.

Discussion
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.

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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

TextOrder

Compares two text strings, taking into account the script and language for each of the strings. (Deprecated in Mac OS X v10.4. Use `CFStringCompare` or `UCCompareText` instead.)

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

Parameters

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 length, in bytes, of the first text string.

bLen

The length, in bytes, of 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.

Return Value

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.

Discussion
This function 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.

"Implicit Language Codes" (page 25) are listed in the Constants section. Most applications specify the language code `scriptCurLang` for both the `aLang` and `bLang` values.

`TextOrder` first calls `ScriptOrder` (page 50); 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` (page 43); 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, it uses the sorting rules specified by the system script or the font script, depending on the state of the international resources selection flag.

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` (page 31) function, 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.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`StringCompare.h`

TypeSelectClear

Clears the key list and resets the type select record. (Deprecated in Mac OS X v10.4. Use `UCTypeSelectFlushSelectorData` instead.)

Not recommended.

```
void TypeSelectClear (
   TypeSelectRecord *tsr
);
```

Parameters

tsr

A pointer to the type-select record you want to clear.

Discussion
The use of this function is not recommended in a Unicode-based application. If you want to use this function in an application that uses the Unicode character set, you must first convert Unicode text strings to Macintosh encoded Pascal text strings. You must also provide the encoding type or be able to determine it by extracting it from the text or by examining the system or keyboard script.

Special Considerations

For Unicode-based applications, you should use the UCTypeSelect functions, which manipulate a `UCTypeSelectRef` object. For more detalis, see Unicode Utilities (`UnicodeUtilities.h`).

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
`TypeSelect.h`

TypeSelectCompare

Compares a text buffer to the keystroke buffer. (Deprecated in Mac OS X v10.4. Use `UCTypeSelectCompare` instead.)

Not recommended.

```
short TypeSelectCompare (
   const TypeSelectRecord *tsr,
   ScriptCode testStringScript,
   StringPtr testStringPtr
);
```

Parameters

`tsr`
A type select record that contains the keystroke buffer.

`testStringScript`
The script code of the test string.

`testStringPtr`
A pointer to the text you want to compare to the keystroke buffer.

Return Value
A numerical value that represents the ordering of the characters in the keystroke buffer with respect to the test string buffer. The value -1 is returned if characters in the keystroke buffer sort before those in `testStringPtr`; 0 if characters in the keystroke buffer are the same as those in `testStringPtr`, and 1 if the characters in the keystroke buffer sort after those in `testStringPtr`.

Discussion
The use of this function is not recommended in a Unicode-based application. If you want to use this function in an application that uses the Unicode character set, you must first convert Unicode text strings to Macintosh encoded Pascal text strings. You must also provide the encoding type or be able to determine it by extracting it from the text or by examining the system or keyboard script.

Special Considerations

For Unicode-based applications, you should use the UCTypeSelect functions, which manipulate a `UCTypeSelectRef` object. For more detalis, see Unicode Utilities (`UnicodeUtilities.h`).

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
TypeSelect.h

TypeSelectFindItem

Finds the closest match between a specified list of characters and the keystrokes stored in the type select record. (Deprecated in Mac OS X v10.4. Use `UCTypeSelectFindItem` instead.)

Not recommended.

```
short TypeSelectFindItem (
   const TypeSelectRecord *tsr,
   short listSize,
   TSCode selectMode,
   IndexToStringUPP getStringProc,
   void *yourDataPtr
);
```

Parameters

tsr

A pointer to the type select record that contains the keystrokes you want to compare.

listSize

The size of the list to search through.

selectMode

The select mode. See `Type Select Modes` (page 26)for a list of the constants you can supply.

getStringProc

A pointer to your index-to-string callback function. See `IndexToStringProcPtr` (page 15) for more information.

yourDataPtr

A pointer to your data structure. This is passed to your index-to-string callback, and can be `NULL`, depending on how you implement your callback function.

Return Value

Discussion
The use of this function is not recommended in a Unicode-based application. If you want to use this function in an application that uses the Unicode character set, you must first convert Unicode text strings to Macintosh encoded Pascal text strings. You must also provide the encoding type or be able to determine it by extracting it from the text or by examining the system or keyboard script.

Special Considerations

For Unicode-based applications, you should use the UCTypeSelect functions, which manipulate a `UCTypeSelectRef` object. For more detalis, see Unicode Utilities (`UnicodeUtilities.h`).

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
TypeSelect.h

TypeSelectNewKey

Creates a new type select record. (Deprecated in Mac OS X v10.4. Use `UCTypeSelectCreateSelector` instead.)

Not recommended.

```
Boolean TypeSelectNewKey (
   const EventRecord *theEvent,
   TypeSelectRecord *tsr
);
```

Parameters

theEvent

A pointer to an event record.

tsr

A pointer to a type select record.

Return Value

Returns `true` if the function executed successfully; `false` otherwise.

Discussion

The use of this function is not recommended in a Unicode-based application. If you want to use this function in an application that uses the Unicode character set, you must first convert Unicode text strings to Macintosh encoded Pascal text strings. You must also provide the encoding type or be able to determine it by extracting it from the text or by examining the system or keyboard script.

Special Considerations

For Unicode-based applications, you should use the UCTypeSelect functions, which manipulate a `UCTypeSelectRef` object. For more detalis, see Unicode Utilities (`UnicodeUtilities.h`).

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TypeSelect.h`

UppercaseStripDiacritics

Converts any lowercase characters in a text string into their uppercase equivalents and strips any diacritical marks from the text. (Deprecated in Mac OS X v10.4. Use `CFStringTransform` instead.)

```
void UppercaseStripDiacritics (
   Ptr textPtr,
   short len,
   ScriptCode script
);
```

Parameters

textPtr

A pointer to the text string to be converted.

len

The length, in bytes, of the text string. The text string can be up to 32 KB in length.

script

The script code of the script system whose resources are used to determine the results of converting characters.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the script specified by the value of the `script` parameter. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

Discussion

`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 function combines the effects of the `UppercaseText` (page 62) and `StripDiacritics` (page 56) functions.

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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

UppercaseText

Converts any lowercase characters in a text string into their uppercase equivalents. (Deprecated in Mac OS X v10.4. Use `CFStringUppercase` instead.)

```
void UppercaseText (
   Ptr textPtr,
   short len,
   ScriptCode script
);
```

Parameters

textPtr

A pointer to the text string to be converted.

len

The length, in bytes, of the text string. The text string can be up to 32 KB in length.

script

The script code of the script system whose case conversion rules are used for determining uppercase character equivalents.

The conversion uses tables in the string-manipulation (`'itl2'`) resource of the specified `script`. You can specify `smSystemScript` to use the system script and `smCurrentScript` to use the script of the current font in the current graphics port.

Discussion

`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.

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.

Special Considerations

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

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In

`TextUtils.h`

UpperString

Converts any lowercase letters in a Pascal string to their uppercase equivalents, using the Macintosh file system rules. (Deprecated in Mac OS X v10.4. Use `CFStringUppercase` instead.)

```
void UpperString (
   Str255 theString,
   Boolean diacSensitive
);
```

Parameters

theString

On input, this is the Pascal string to be converted. On output, this contains the string resulting from the conversion. `UpperString` traverses the characters in `theString` and converts any lowercase characters with character codes in the range 0x00 through 0xD8 into their uppercase equivalents. `UpperString` places the converted characters in `theString`.

diacSensitive

A flag that indicates whether the case conversion is to strip diacritical marks. If the value of this parameter is `TRUE`, diacritical marks are considered in the conversion; if it is `FALSE`, any diacritical marks are stripped.

Discussion

Only a subset of the Roman character set (character codes with values through $D8) are converted. Use this function to emulate the behavior of the Macintosh file system.

Availability

Available in Mac OS X v10.0 and later.

Deprecated in Mac OS X v10.4.

Not available to 64-bit applications.

Declared In
TextUtils.h

upperstring

Converts any lowercase letters in a Pascal string to their uppercase equivalents. (Deprecated in Mac OS X v10.4. Use `CFStringUppercase` instead.)

Not recommended

```
void upperstring (
   char *theString,
   Boolean diacSensitive
);
```

Parameters

theString

On input, this is the Pascal string to be converted. On output, this contains the string resulting from the conversion.

diacSensitive

A flag that indicates whether the case conversion is to strip diacritical marks. If the value of this parameter is `TRUE`, diacritical marks are considered in the conversion; if it is `FALSE`, any diacritical marks are stripped.

Discussion
You should use the function `CFStringUppercase` instead of this one.

Carbon Porting Notes

Use `UpperString` instead.

Availability
Available in Mac OS X v10.0 and later.
Deprecated in Mac OS X v10.4.
Not available to 64-bit applications.

Declared In
TextUtils.h

Unsupported Functions

This section lists functions that are unsupported and you should no longer use. Table B-1 provides information on what you should do in place of using these functions.

Table B-1 Porting notes for unsupported functions

Unsupported functions	Porting notes
`equalstring`	Use `CFStringCompare` instead.
`FindWord`	Use `UCFindTextBreak` instead.
`IUCompPString`	Use `CFStringCompare` instead.
`iucomppstring`	Use `CFStringCompare` instead.
`IUCompString`	Use `CFStringCompare` instead.
`iucompstring`	Use `CFStringCompare` instead.
`IUEqualPString`	Use `CFStringCompare` instead.
`iuequalpstring`	Use `CFStringCompare` instead.
`IUEqualString`	Use `CFStringCompare` instead.
`iuequalstring`	Use `CFStringCompare` instead.
`IULangOrder`	Use `CFStringCompare` or `UCCompareText` instead.
`IUMagIDPString`	Use `CFStringCompare` instead.
`IUMagIDString`	Use `CFStringCompare` instead.
`IUMagPString`	Use `CFStringCompare` instead.
`IUMagString`	Use `CFStringCompare` instead.
`IUScriptOrder`	Use `CFStringCompare` or `UCCompareText` instead.
`IUStringOrder`	Use `CFStringCompare` or `UCCompareText` instead.
`iustringorder`	Use `CFStringCompare` or `UCCompareText` instead.
`IUTextOrder`	Use `CFStringCompare` or `UCCompareText` instead.
`LowerText`	Use `CFStringLowercase` instead.
`LwrText`	Use `CFStringLowercase` instead.

Unsupported functions	Porting notes
`newstring`	Use `CFStringCreateCopy` instead.
`NFindWord`	Use `UCFindTextBreak` instead.
`numtostring`	Use `CFStringCreateWithFormat` instead.
`setstring`	Use `CFStringCreateWithPascalString` and `CFStringReplaceAll`.
`stringtonum`	Use `CFStringGetIntValue` instead.
`StripText`	Use `CFStringTransform` instead.
`StripUpperText`	Use `CFStringTransform` instead.
`UpperText`	Use `CFStringUppercase` instead.
`UprText`	Use `CFStringUppercase` instead.
`getindstring`	Use `CFBundleCopyLocalizedString` instead.

Document Revision History

This table describes the changes to *Text Utilities Reference*.

Date	Notes
2007-05-29	Made minor formatting changes.
2006-07-24	Added deprecation information.
2003-02-12	Updated formatting. Added documentation for numerous data types and constants. Added documentation for type-select functions.

Document Revision History

Index

L

M

N

O

P

R

S

T

U